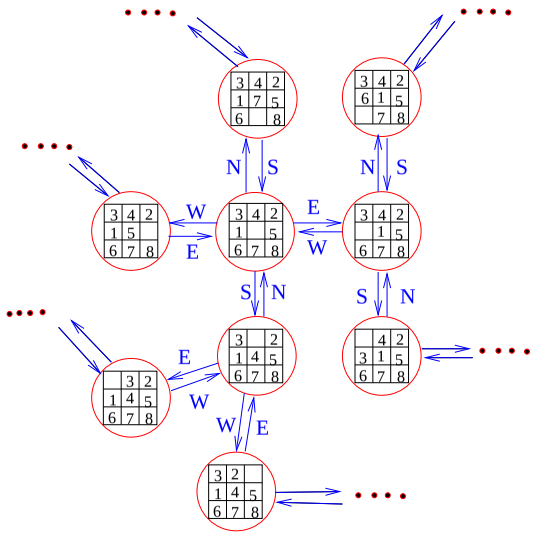
Data Structures

Goal: Understand data structures by solving the puzzle problem

- Elementary Structures
  1. Arrays
  2. Lists
  3. Trees
- Search Structures
  1. Binary search trees
  2. Hash tables
- Sequence Structures
  1. Stacks
  2. Queues
  3. Priority queues
- Graphs

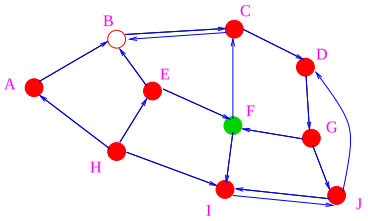
A Motivating Application

State Transition Graph of 8-Puzzle



Graph: a very general data structure

V: set of nodes E: set of edges (pairs of nodes)



V = {A,B,C,D,E,F,G,H,I,J}

E = {(A,B), (B,C),(C,B),.....}

Edge (A,B):

A is source of edge

B is destination of edge

Graphs can represent state transitions, road maps, mazes .....

In some graphs, edges may have additional information.

- puzzle graph: edges annotated with N/S/E/W

In some graphs, certain nodes may be special

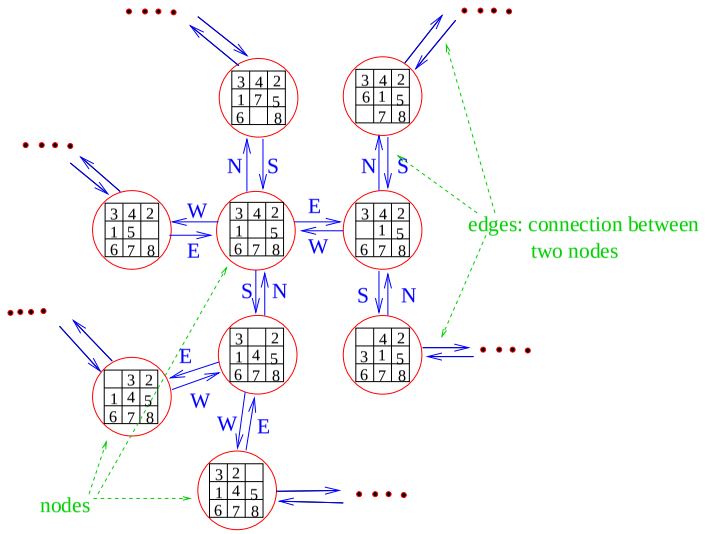
- puzzle graph: initial and final (sorted) nodes are special

Graph in example is a DIRECTED graph

Undirected graph: no arrows on edges

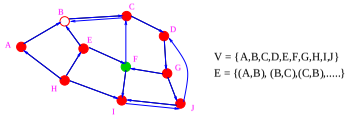
analogy: 1-way street vs 2-way street

Graph: set of Nodes and Edges between nodes



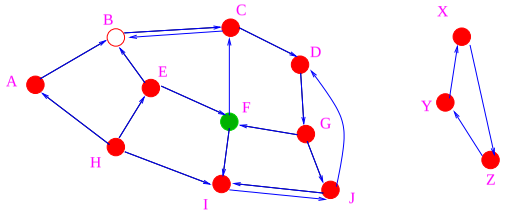
- Path: a sequence of edges in which destination node of an edge in sequence is source node of next edge in sequence
  Examples:(i) (A,B),(B,C),(C,D) (ii) (H,I),(I,J)
- Source of a path: source of first edge on path
  Destination of path: destination of last edge on path
- Reachability: node n is said to be reachable from node m if there is a path from m to n.
- There may be many paths from one node to another.
  Example: (E,F) and (E,B),(B,C),(C,D),(D,G),(G,F)
- Simple path: a path in which every node is the source and destination of at most two edges on the path
  Example: (not a simple path) (C,B),(B,C),(C,D)
- Cycle: a simple path whose source and destination nodes are the same. Example: (i) (C,B),(B,C) (ii) (D,G),(G,J),(J,D)

Some terminology



- Out-edges of a node n: set of edges whose source is node n (eg. out-edges of C are {(C,B),(C,D)})
- Out-degree of a node n: number of out-edges of node n
- In-edges of a node n: set of edges whose destination is node n (eg. in-edges of C are {(B,C),(F,C)})
- In-degree of a node n: number of in-edges of node n
- Degree of a node n in an undirected graph: number of edges attached to n in that graph
- Adjancency: node n is said to be adjacent to node m if (m,n) is an edge in the graph
  Intuitively, we can get from m to n by traversing one edge.

Path problems



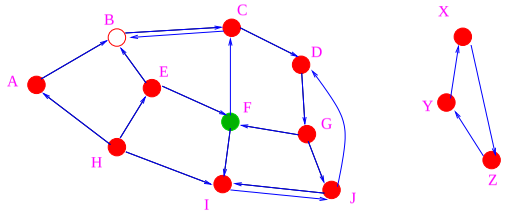
Length of a path: number of edges in path

Minimal path problems:
Find the shortest path from node A to node F.
Find the shortest path from every node to F.

For puzzle problem, this corresponds to path with fewest moves.

Sometimes, edges have distances.
Length of path = sum of lengths of distances on path.
This is more appropriate for path problems in graphs representing maps.

Path problems



Many interesting problems can be phrased as path problems in graphs.

(1) Is there a way to reach the sorted state, starting from any scrambled position of tiles?

Reachability problem:
Is there a path from every node to the node representing sorted position?

For puzzle problem, answer is no!

Sam Loyd: cannot reach sorted state from

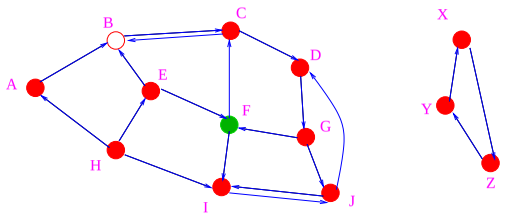
1	2	3
4	5	6
8	7	0

These kinds of problems are studied in *graph theory*.

If you get turned on by this stuff, become a CS major and take CS 410, 381/481 etc.

We will have time only to study some simple problems like reachability, with the goal of understanding modern data structures.

Path problems



Cycle: Path that starts and ends at same node.

Travelling salesman's problem:
Find the smallest length cycle that passes through all nodes.

Easy to come up with slow algorithms for this problem.
No one knows if there is an efficient algorithm for this problem.
(NP/NP-complete problems)

Requirements:

1. should not get stuck in cycles (correctness)
2. should be exhaustive: if we terminate without reaching sorted state, sorted state must be unreachable from scrambled state (correctness)
3. should not repeatedly examine states adjacent to a state(efficiency)

Goal: write a program to determine if sorted state is reachable from scrambled state for puzzle problem

Idea:

- Start in the scrambled state.
- Generate states adjacent to scrambled state.
- Generate states adjacent to those states.
  ....
- Stop if you either generate sorted state or you have generated all states reachable from scrambled state.

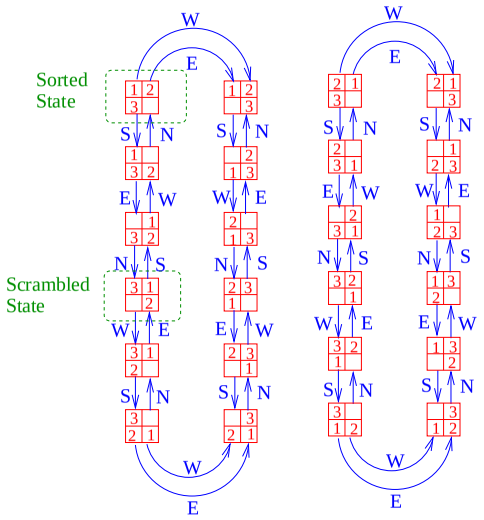
Key Idea: Keep two sets of nodes

1. Visited : set of configurations we have generated at least once
2. Frontier : subset of Visited nodes whose adjacencies we have not yet examined

Pseudocode for Graph Search algorithm:

```
initialize Frontier set and Visited set with scrambled configuration;
while (Frontier set is not empty)
  {Remove a node  v  from Frontier set;
   for each node w adjacent to v do //there is edge (v -> w)
       {If  w  is the goal node, declare victory;
        Otherwise, check if  w  is in Visited set;
        If not, add it to Visited and Frontier sets;
       }
  }
```


Let us try to execute a few steps of the algorithm for this problem.

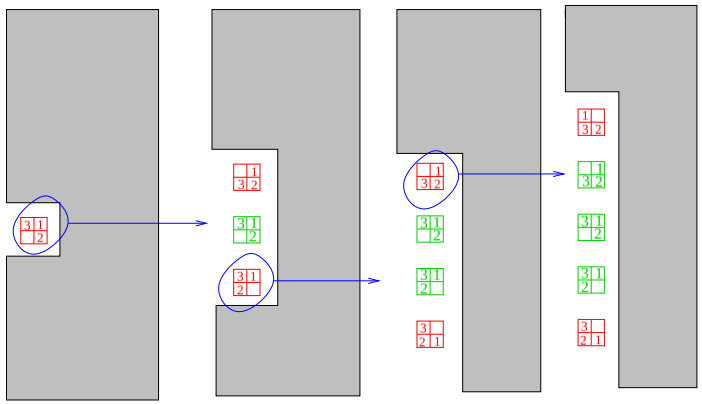


To make this a program, we need to answer the following questions:

(1) SEQUENCE STRUCTURE: In what order should we get nodes from the *Frontier* set? What data structure can we design to give us the nodes in that order? Answer: stacks, queues, priority queues

(2) SEARCH STRUCTURE: How do we organize the *Visited* set so that we can search it efficiently? Answer: arrays,lists, binary search trees, hash tables,....

Algorithm for Solving Puzzle : some possible initial steps



: configuration in Frontier set and Visited set

: configuration only in Visited set

Note: we are not keeping track of moves that brought us to a given configuration

Writing generic code:

- Order in which we explore nodes (order in which they are removed from Frontier set) is very important, and can make a big difference in how quickly we find solution.

  How can we write code so that it works for any sequence structure? Answer: use subtyping

- Most time-consuming part: searching if node is in Visited set.

  How do we write code so that it works for any search structure? Answer: use subtyping

- Graph search algorithm works for any graph, not just puzzle state transition graph (all we need to some way to determine what nodes are adjacent to a given node).

  How can we write code so that it works for any graph? Hmm - need Iterators/Visitors... See later.

Two key interfaces:

SeqStructure: all sequence structures implement this interface

SearchStructure: all search structures implement this interface

For search structure, fast search is important!

For sequence structure, fast lookup is not important!

An interface IPuzzle for puzzles and a class called ArrayPuzzle are defined in code.

Understanding Sequence Structures

Code for Simulating Puzzle

See code at end of handout.

The code we have written will work for any search structure and sequence structure that implement the interfaces defined before. Subtyping is wonderful!

The puzzle state transition graph is hardwired into the code. We cannot use it to perform a graph search in a general graph. We will fix this later.

How do we implement a good sequence structure?

How do we implement a good search structure?

Implementing sequence structures

Key operations:

- put
- get

Question: How are puts and gets related?
=>
What order should we explore the nodes in the graph?

Popular graph search strategies:

1. heuristic graph search
2. oblivious graph search
   - breadth-first graph search
   - depth-first graph search

Heuristic graph search

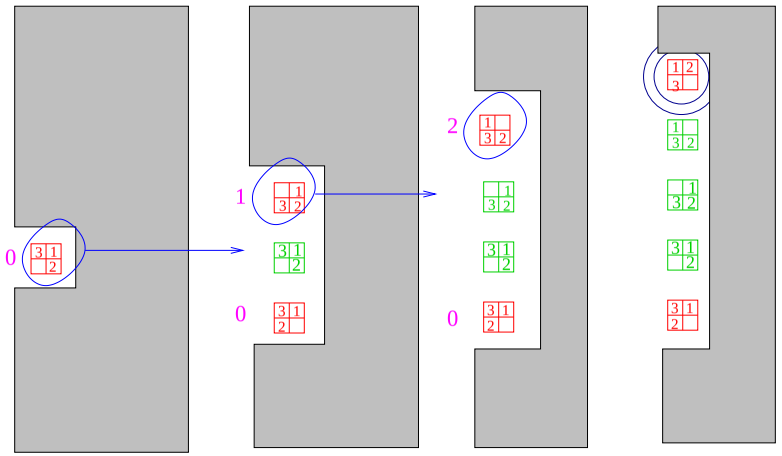
In many applications, we have some domain-specific information that we can exploit to "guide" the graph search.

Example: for Puzzle, from all configurations in Sequence Structure, pick the one that has

1. largest number of tiles in correct position, or
2. minimum sum of Manhattan distances of tiles from their correct positions, or
3. .....

These are heuristics.

Heuristic: pick configuration with greatest number of correctly placed tiles



Number in pink = "figure of merit" for configuration

Caveat: This heuristic is good only when you are close to sorted state....

How do we implement this heuristic graph search in our approach?

Priority Queue

- Each item in Sequence Structure has an associated integer value called its *priority*.
- Method `get()` should always return the entry with the largest priority (in some designs, lowest priority).
- If there are multiple items with highest priority, return the one that was put earliest.

In our example,
Priority of configuration = number of correctly placed tiles

How do we implement priority queues?

Oblivious Graph Search

Sometimes, you may not have heuristics to guide your graph search.

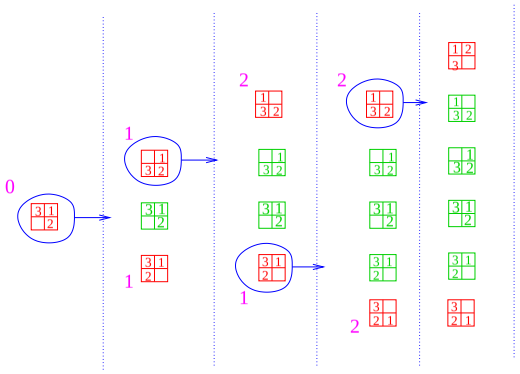
Oblivious graph search: graph search strategy is dictated by structure of graph and not by heuristics

Two most popular strategies:

- breadth-first graph search
- depth-first graph search

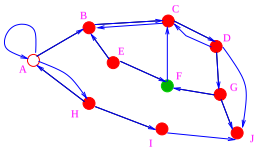
Breadth-first search

Sequence structure returns configuration from oldest live generation.



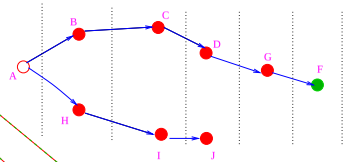
0,1,2...: these are generation numbers for configurations

Breadth-First Search (given start node): gerontocracy



Generation i = set of all nodes n such that shortest path from start to n has i edges

Generation 0: {A}
Generation 1: {B,H}
Generation 2: {C,I}
Generation 3: {D,J}
Generation 4: {G}
Generation 5: {F}



Breadth-first search: explore nodes generation by generation

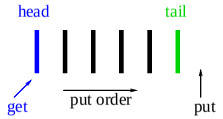
✓ A,B,H,C,I,D,J,G,F

~~A,B,C,D,G,J,I,H,F~~ not BFS

✓ A,H,B,I,C,D,J,G,F

Queues show up in many applications where the service discipline is first-in-first-out (FIFO)

- requests for service: SP-2 jobs
- simulations of systems like bank teller machines, public transportation etc. where service discipline is FIFO
- circuit simulations: devices are simulated in time order



In the context of queues,

put is called enqueue

get is called dequeue

How do we implement sequence structure for BFS?

One approach: use priority queue

- priority = generation number
- priority queue returns lowest priority entry

Another approach: use a sequence structure that obeys First-in-First-out discipline

Queue: get returns item that was put earliest.

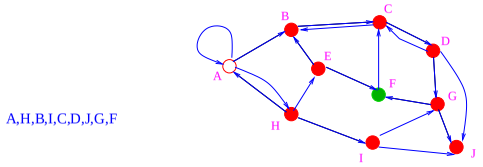
Why use a BFS?

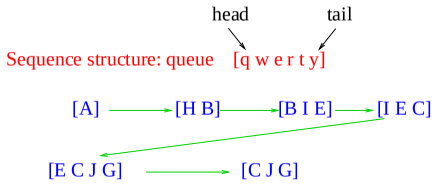
It is easy to show that BFS can determine the shortest path (smallest number of moves) from scrambled configuration to sorted configuration.

Remember: in general, there may be many paths from a node to another node in the graph. Heuristic graph search may take the long way home.

Why use heuristic graph search then?

Queue: First-in-first-out Sequence Structure



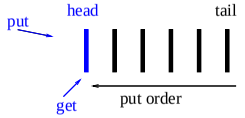


Queues are simpler to implement than priority queues.
Queues are faster than priority queues.

=> We will design special structures for queues rather than reuse priority queues.

Another oblivious graph search strategy: use a stack

Node order given by sequence structure that is
last-in-first-out (LIFO) (aka stack as in stack of coins)

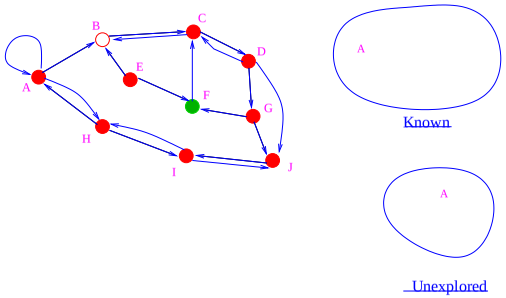


In the context of stacks,

put is known as push

get is known as pop

Example:



A stack is a LIFO sequence structure.

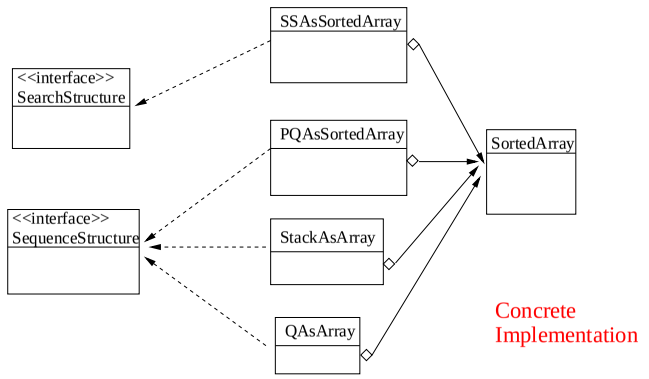
This graph search strategy is related to depth-first graph search.

Implementing Sequence and Search Structures Using Sorted Arrays

Arrays are not very appropriate for our purpose because our data structures grow and shrink.

However, we will start with arrays to get a feel for search and sequence structure.

Bridge classes: decoupling abstractions from implementations



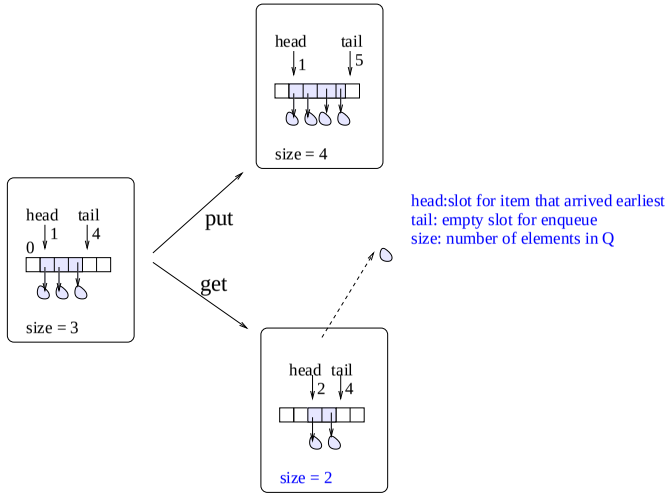
Using arrays to implement a Search Structure

- Keep items in an array, and track number of items in array.
- To locate an item l in the array, search the array using binary search. If you find an item that is equal to item l, return true; otherwise return false. Time: $O(log(n))$.
- To insert a new item to the structure, search the array as above. If you find the item is already there, there is nothing to do. Otherwise, if the search procedure says the item ought to be in index i, shuffle all items in array from index i onwards one slot to the right to make room for the new item, and stick the item into index i. Time: $O(n)$
- Deletion is similar: do a search. If item is not in array, there is nothing to do. Otherwise, if item is in index i, shuffle all items from index i+1 onwards one slot to the left to squeeze out the item to be deleted. Time: $O(n)$

Using Arrays to implement Sequence Structures

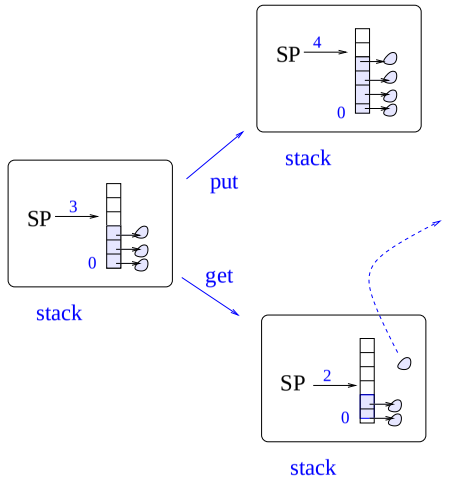
See code in SSAsSortedArray.

Implementing Queues using arrays



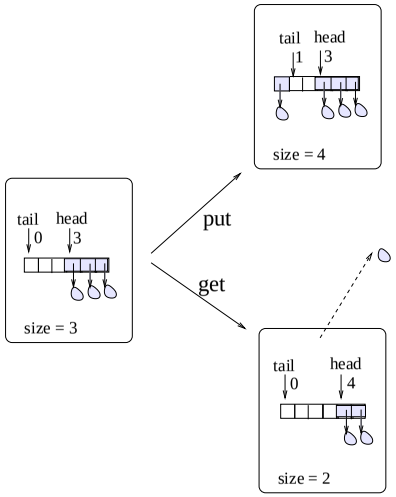
Use the array as a "circular buffer".

Using arrays to implement stacks



See class StackAsArray at end of handout.

Wrap-around: array is a ring buffer!



Exercise: can you compute size from values of head and tail?

Check: does your expression work for empty Q? full Q?

Implementing priority queues using arrays.

Key question: how should be the type of priority queue elements?

Possibilities:

- Objects containing a puzzle and an integer:

```
class PQElement {
 IPuzzle p;
 int priority;
 ... }
```

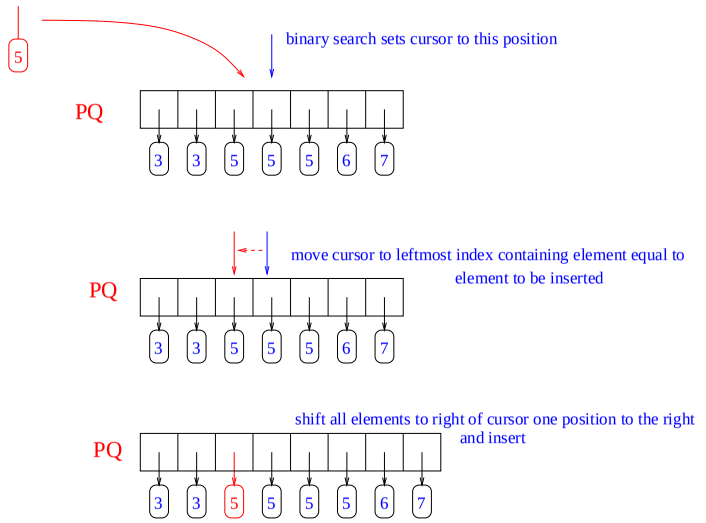
  Criticism: this class is too specific to the puzzle problem.

- More generic solution: Objects containing an object and an integer

```
class PQElement {
 Object p;
 int priority;
 ... }
```


See code in class QAsArray.

Small detail about PQ Insertion

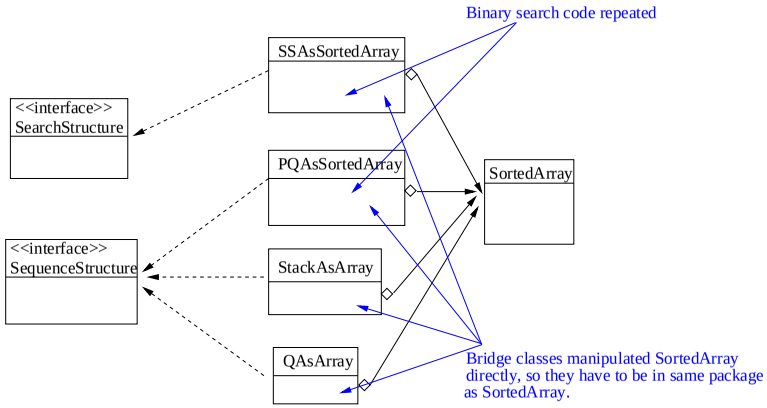


See class PQAsSortedArray at end of handout.

Problem with arrays: fixed size

- Sequence structures grow and shrink as put() and get() are done.
- If we use an array, we must create an array big enough to hold maximum number of elements that could ever exist at one time in structure (or copy elements from small arrays to bigger arrays).
- Insertion/deletion may require shuffling elements to eliminate "holes" in the array.
- Can we use "dynamic" data structures that grow on demand?
- One solution: use linked lists

Critique of our implementation



Question: Can we define a suitable abstraction of SortedArray that can be used by Bridge classes like SSAsSortedArray?

This leads to the idea of Collections. See later.

Lists

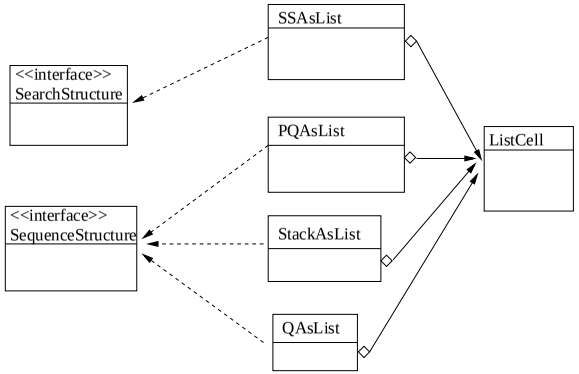
Advantage of lists: can grow and shrink on demand

Disadvantage of lists: no random access

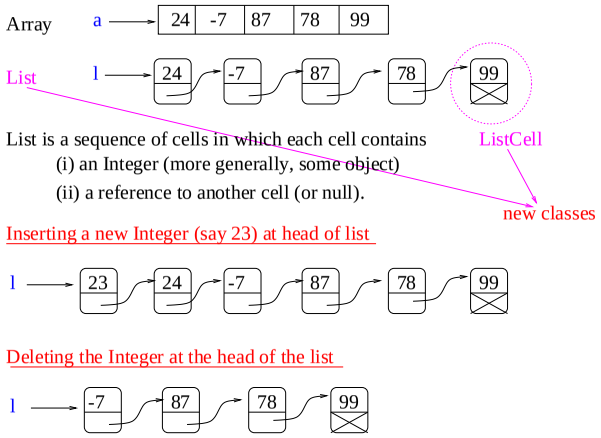
Lists are fine for implementing stacks and queues since they do not require random access.

Lists are no so great for search structures and priority queues. Solution: use trees. (see later)

Implementation using Lists



Linked Lists



List is a sequence of cells in which each cell contains
(i) an Integer (more generally, some object)
(ii) a reference to another cell (or null).

Inserting a new Integer (say 23) at head of list

Deleting the Integer at the head of the list

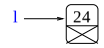
In general, we may want to insert and delete from places other than the head of the list.

```
  //this is sometimes called the "rplaca" method
  public void setDatum(Object o) {
    datum = o;
  }
  //this is sometimes called the "rplacd" method
  public void setNext(ListCell l){
    next = l;
  }
}
```

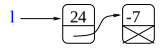

ListCell class

```
class ListCell {
 protected Object datum;
 protected ListCell next;

  public ListCell(Object o, ListCell n){
    datum = o;
    next = n;
  }
  //this is sometimes called the "car" method
  public Object getDatum() {
    return datum;
  }
  //this is sometimes called the "cdr" method
  public ListCell getNext(){
    return next;
  }
```

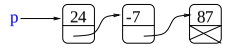

Examples of list manipulation: (assume you can store int's in ListCell)



l = new ListCell(24,null);



l = new ListCell(24, new ListCell(-7,null));



p = new ListCell(24,new ListCell(-7,new ListCell(87,null)));

Another way:

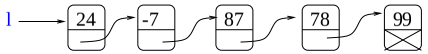
p = new ListCell(24, new ListCell(-7,null));

p.getNext().setNext(new ListCell(87,null));

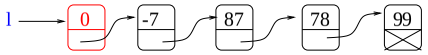
Another way:

p = new ListCell(87,null);
p = new ListCell(-7,p);
p = new ListCell(24,p);

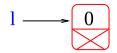
Make sure you understand all this code.

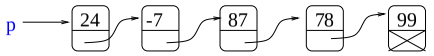


l.setDatum(0);

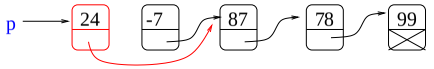


l.setNext(null);



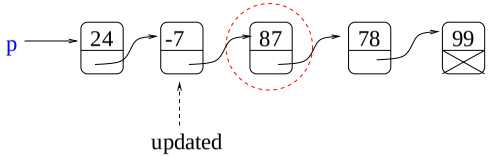


p.setNext(p.getNext().getNext());



Why? Note that p.getNext().getNext() gives you a reference to the cell containing 87. This statement has the effect of splicing out the cell containing -7.

```
public static void deleteLast(ListCell l) {

  ListCell finger,previous; //previous is always one cell behind finger
  //check that list has at least two elements
  if (l == null) return; //empty list
  if (l.getNext() == null) return; //one element list
  previous = l;
  finger = l.getNext();
  while(finger.getNext() != null) {//while finger is not at last element
    previous = finger;
    finger = finger.getNext();
  }
  previous.setNext(null);
}
```

How do we delete the cell containing 87?

ListCell updated = p.getNext();
updated.setNext(updated.getNext().getNext());

How do we delete last cell of any list?

Need to write a loop that walks over list.
When it gets to second to last element of list,
it sets its Next field to null.
How do we write this code?
See class deleteLast.

Implementing queues using linked lists

Head of list: earliest entry

Tail of list: last entry

Complication: put and get work at opposite ends of the list.

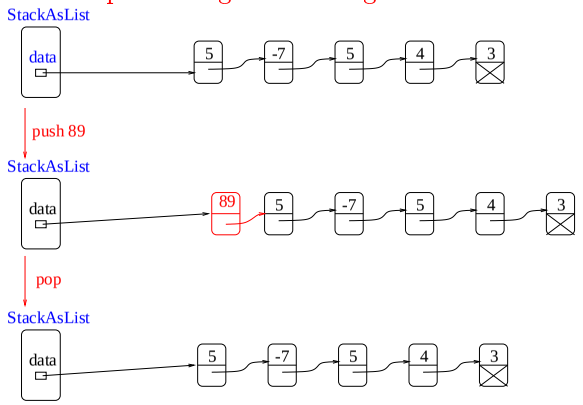
One solution:

- perform gets from head of list
- to do a put, walk down the list till you get to the last cell, and then update this cell to point to the cell you are inserting
- better solution: keep track of last ListCell in list

See class QAsList.

$O(1)$ complexity for put and get.

Implementing stacks using linked lists



See class StackAsList for an implementation.

$O(1)$ complexity for put and get.

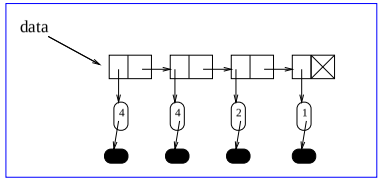
Designing data structures for PriorityQ's

Keep PQ items in decreasing order of priority
(opposite order from PQAsSortedArray) (why?)

Entries with same priority are in FIFO order.

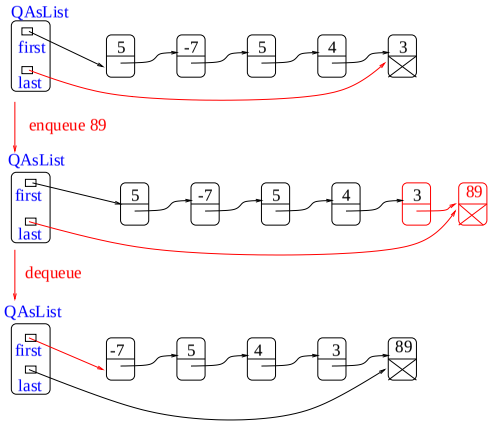
put: walk down list and insert into "right place"

get: extract from head of list



See class PQAsList

$O(n)$ put time, $O(1)$ get time.



Lists can also be used to implement Search Structures

```
interface SearchStructure {
  void insert(Object o); //stick into search structure
  void delete(Object o); //remove objects equal to o from search structure
  boolean search(Object o);
  int size();
 }
```

We must do a linear search - no random access!!

So maintain entries in search structure as an unsorted list.

Intuitive idea:

- `insert`: stick into head of list if element not in list($O(n)$ time)
- `search`: walk down list till you find it (maybe)($O(n)$ time)
- `delete`: look for item and splice it out ($O(n)$ time)

See class SSAsList.

Important special case of priority queues:

fixed number of priorities (say 0..p-1)

Example: heuristic search with # of out-of-place tiles = 0..9

Cool implementation of priority queue for this case:

1. Use an array of p Queues (one for each priority level)
2. Implement put by enqueuing into queue for the appropriate priority
3. Implement get by searching for non-empty Queue with highest priority elements, and dequeue from that Queue.

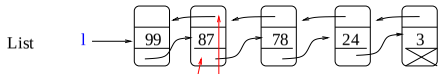
$O(1)$ put time, $O(p)$ get time where p is number of priority levels.

Summary of lists

- Lists can grow and shrink as needed.
- Price: no random access, so only linear search.
- Insertion and deletion from lists: easy, just remember to keep track of *previous* list cell.
- Stack, queue: $O(1)$ put and get time
- PQ, Search Structure: $O(n)$ time for most operations. Heart of problem: no random access.

One solution: trees.

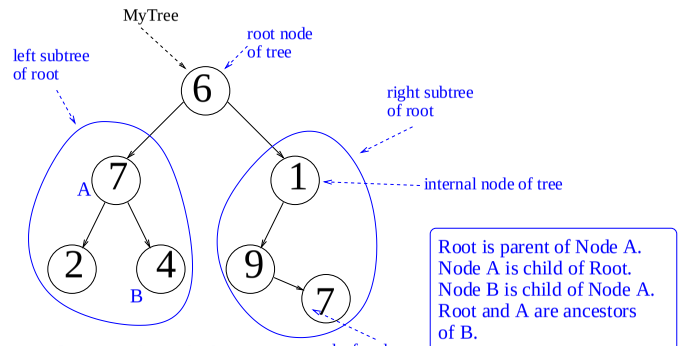
Some applications require traversing list both forward and backward.
Solution: doubly-linked lists (dLLists)




```
class dLCell{
    protected Object datum;
    protected dLCell forward;
    protected dLCell backward;
    ·····//the usual getter and setter methods
    }
```

Good exercise: rewrite PQ and SS code so it uses doubly linked lists, and eliminate the *previous* cursor.

Trees



Root is parent of Node A.
Node A is child of Root.
Node B is child of Node A.
Root and A are ancestors of B.
A and B are descendants of Root.

Rooted tree:graph in which
- there is a distinguished node called the root
- root has no incoming edges
- all other nodes have one incoming edge
- there is a path from root to every node

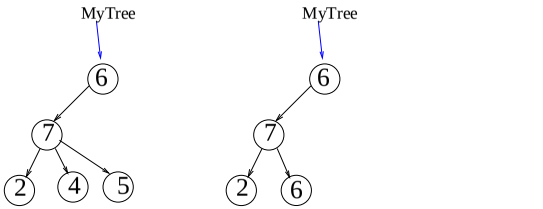
Depth of a node = length of simple path from root to that node
depth(A) = 1 depth(B) = 2

Height of a node = length of longest simple path from node to a leaf
height(A) = 1 height(B) = 0
Height of tree = Height of root (in our example, 3)

Cell for building binary trees:

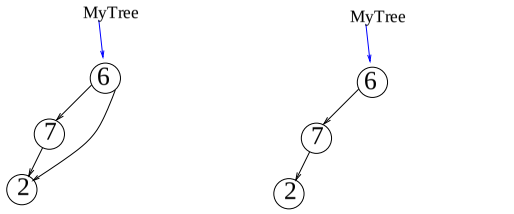
```
//declaration of tree cell
class TreeCell {
  protected Object o;
  protected TreeCell left;
  protected TreeCell right;

  public TreeCell(Object i) {
    o = i; //left and right are null by default
  }
  public TreeCell (Object i, TreeCell l, TreeCell r) {
    o = i;
    left = l;
    right = r;
  }
  public void setObject(Object o) {
    this.o = o;
  }
  public Object getObject() {
    return o;
  }
  public void setLeft(TreeCell t) {
    this.left = t;
  }
  public TreeCell getLeft() {
    return left;
  }
  public void setRight(TreeCell t) {
    this.right = t;
  }
  public TreeCell getRight() {
```


Some Trees



(i) General Tree

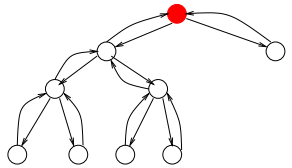
(ii) Binary tree: every node has at most two children



(iii) Not a tree

(iv) Special case of binary tree:list-like tree!

In some applications, it is useful to have trees in which nodes point back to their parents (analog of doubly linked lists):

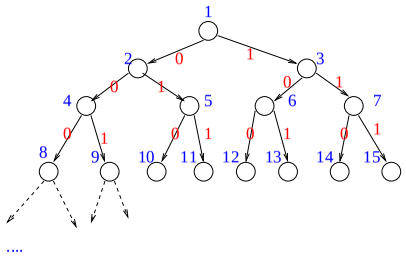


```
class DLTreeCell extends TreeCell{
  protected TreeCell parent;
  ....//getter and setter methods

}
```

This lets you walk from a leaf upto the root.

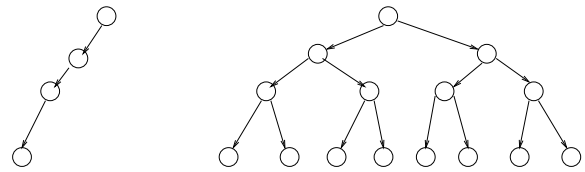

```
    return right;
  }
  public String toString() {
    String lString = "";
    String rString = "";
    if (left != null)
      lString = left.toString();
    if (right != null)
      rString = right.toString();
    return lString + " " + ((Integer)o).intValue() + " " + rString;
  }
}
....
//build the second binary tree in previous slide
TreeCell temp = new TreeCell(new Integer(7),
                             new TreeCell(new Integer(2)),
                             new TreeCell(new Integer(6)));
TreeCell myTree = new TreeCell(new Integer(6), temp, null);
...
```

Standard Numbering Scheme for Binary Tree Nodes



Why is this encoding scheme useful?

Some useful factoids about binary trees



Skinny tree

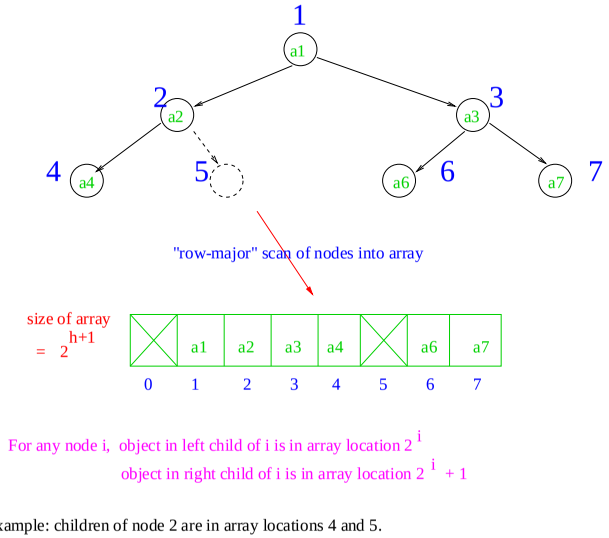
Bushy tree: full binary tree of height 3

Binary tree of height h has at least (h+1) nodes and at most $2^{(h+1)} - 1$ nodes

You can invert this calculation to find minimum and maximum heights for a tree with a given number of nodes: a binary tree with n nodes has a maximum height of $n - 1$ and a minimum height of $\lceil log_2(n + 1) - 1 \rceil$.

Number of leaves in a *full* binary tree with height h $= 2^h$

One use of encoding: store tree into array w/o storing edges.



Example: children of node 2 are in array locations 4 and 5.

Another use of encoding scheme: node number correspond to path from root to that node in a simple way:

Given a node number, write the number in binary notation using as few bits as possible. Drop leading 1 in front of number. The resulting binary number encodes the path from root to this node: 1 means left and 0 means right.

Example: node 6 (in binary, 110). Drop leading 1 to get 10. Path is Right, Left.

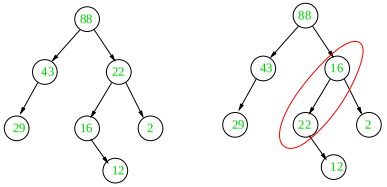
Examples of heaps: ages of people in family tree
Parent is always older than children, but you can have an uncle who is younger than you.

Salaries of people in organization: bosses make more than subordinates, but a 2nd level manager in one sub-divison may make more money than a 1st level manager in a different sub-division

Heap

Tree in which

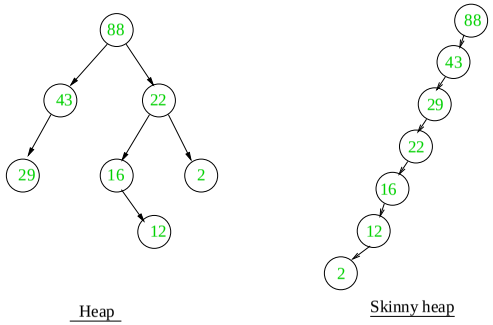
1. integer stored in nodes
2. integer stored in a node is $\geq$ than integers stored in any of its children



Heap Not a heap

Easy to show this means integer at node is $\geq$ than integer in any descendant.

Degenerate case of heap: long and skinny tree (list!)

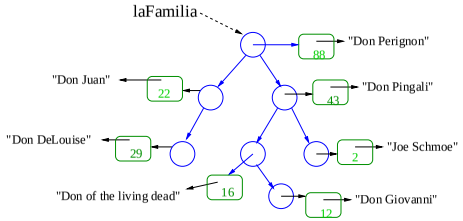


Heap Skinny heap

My running example of heap: crime family

Entries are PQEelements containing a name and an integer = number of murders committed by person (measure of his ruthlessness)

Boss must be more ruthless than subordinates, so crime family is a heap.



Heap of Priority Queue Elements

Let us look at `get` first.

Element to extract is in root of tree (why?).
Removing that element leaves a "hole" in the root.
How should we fill that hole?
One solution:

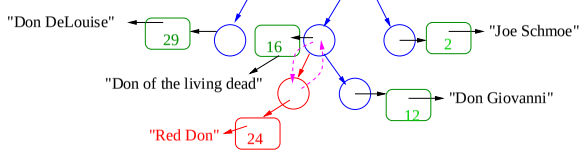
move maximum of children of root to root.
(in our example, "Don Pingali" is moved into root)
Problem: this creates a hole where "Don Pingali" used to be.
Apply same idea again: find maximum of children of "Don Pingali"
("Don of the Living Dead") and move him into "Don Pingali"'s slot
Keep moving elements up till you move a leaf element up.
Remove empty leaf.

Can we implement a priority queue using a heap?

Question: how do we get and put?

Get: extract the element with largest priority

Put: insert element into data structure and preserve heap property



Heapifying after removing root element

Put into a heap

- stick new element into a new leaf node anywhere in tree
- result is not necessarily a heap
- compare parent of new leaf and new leaf and exchange if necessary
  (in our example, we would exchange "Red Don" and "Don of the living dead" since "Red Don" is more ruthless)

- if no exchange was needed, we are done : we have a heap
- otherwise, let p be the parent of the leaf node. We now have to compare p and parent(p) to see if heap condition is violated.

- if so, exchange, etc.
- this process has to terminate at the root of the tree at the worst.

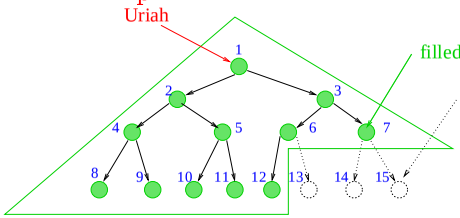
Advantage of heap

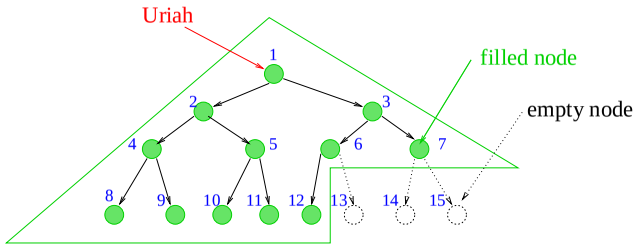
We can implement priority queues using either linked lists or heaps.

Advantage of heap: if tree is not long and skinny, each put and get needs to look only at a small number of elements in the priority queue at any time

Problem with our heap design: there is no guarantee that we will not end up with long and skinny heap (list), so in the worst case, our implementation will not be any more efficient than the list implementation from assignment 5.

More advanced design to ensure tree is fat and short:
ensure heap is a complete binary tree





- "complete binary tree": if node n is occupied, all nodes numbered less than n are also occupied

- if we can maintain a heap as a complete binary tree, tree will be short and bushy. Can we design put and get to "complete binary tree" property is maintained after operation?

- Put: insert new element into "first" empty node (in example, node 13), rather than into any random new leaf node as before, and heapify up the path from this new node to root.

  Problem: how do we know where in tree to create the new node (node 13 in example)? Easy: keep track of size of heap (in example, size is 12). Put will make size = 13, and 13 is 1101, so path to new leaf is RLR.

Get operation that maintains complete tree-ness

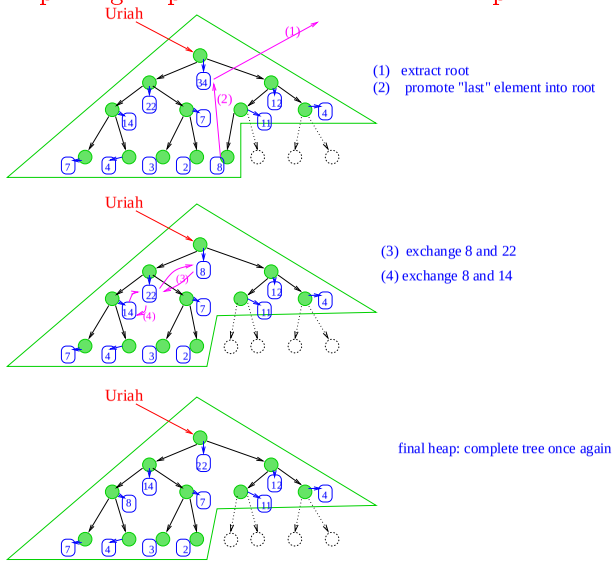
- get algorithm that ensures complete tree-ness:
  - extract element from root of tree and return it
  - in old algorithm for get, we would promote max of nodes 2 and 3 to root, and keep going recursively down the tree. However, this may create a "hole" in some position like 8 or 9 ultimately, and we lose complete tree-ness
  - clever way to fill root: promote element from "last" filled node (in our example, node 12) to root
  - this may violate heap property, so heapify by comparing new root element with elements in 2 and 3, and exchanging root with largest of its children etc. (intuitively, if new root element is a loser, he sinks down the tree till he needs to sink no more)

Summary of priority queue implementation using heaps

- Use a heap that is a complete binary tree to store PQ elements. Keep track of size of PQ.
- Get: return the root element. Promote the "last" element of the complete binary tree to the root position and walk down the tree restoring heap property. Accessing last position: use binary representation of integer size. $O(log(n))$ time.
- Put: insert into "last +1" element of complete binary tree, heapifying as you walk down the tree. $O(log(n))$ time.

See class Heap for code.

Example of get operation that maintains complete tree-ness



Let us now see how to use trees to implement a fast Search Structure.

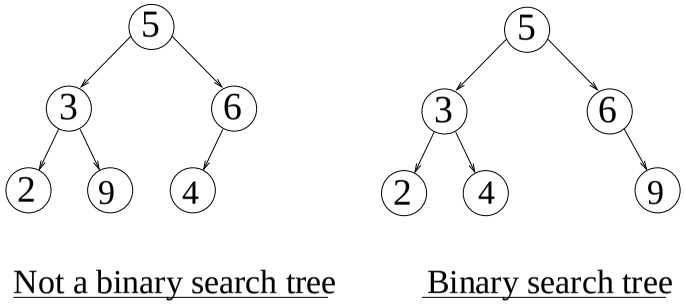
Binary tree: contains integers in some order

Binary search tree: special case of binary tree

At any node n in the tree,

- all integers smaller than integer at node n are stored in the left subtree
- all integers larger than integer at node n are stored in the right subtree

Binary Search Trees



Not a binary search tree

Binary search tree

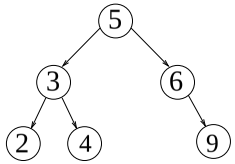
Intuition behind binary search trees:

- start with sorted list
- for efficient search, we want access middle of list
- "pick up" list by the scruff of its neck at some internal element (this will be the root of the tree)
- sub-lists to left and right of this element will flop down
- detach these sub-lists
- repeat process recursively with these sublists, hooking their roots to previous root etc.

Algorithm for searching in binary search tree:

- If tree is empty, return false;
- If ((object at root) = (search object)) return true.
- If ((object at root) < (search object)) search in right subtree
- If ((object at root) > (search object)) search in left subtree.
- While traversing, update cursors properly.

Insertion



Algorithm to insert V into BST:

- Search for V in data structure.
- If V is not there, you'll drop out of BST at some node N.
- Create a new TreeCell T containing V; if V is less than the contents of node N, make T the left child of N; otherwise, make it the right child of N.

Helper function `extractMax` : remove largest element in tree

Algorithm:

- Traverse Right tree edges till you reach node (n) for which Right = null
- Value stored at this node n is maximum. Delete this node, and make left subtree of n the right subtree of parent of n.

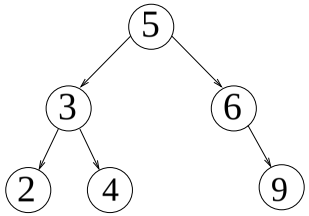
Algorithm for deletion: delete integer i

- Walk down tree till you find node N that contains i.
- Let p be the parent node of N.
- If left subtree of N is empty, make right subtree of N into subtree of p.
- If left subtree of N is not empty, extract maximum value from left subtree of N and stick that into N.

This works, but a more elaborate algorithm might also look to see if right subtree of N is empty before going to extract max from the left subtree.

Intuition behind algorithm: think of tree as a representation of sorted list obtained by picking up list by scruff of its neck.

Deletion Example



See class BST for code for search/insertion/deletion in binary search trees.

Unfortunately, our trees are not necessarily balanced!

This means search in our bst can sometimes take as long as search in a list!

If tree is balanced, search becomes much more efficient.

Self-balancing Trees

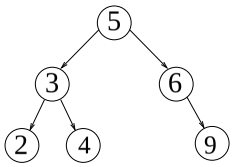
Large body of research on how to 'update' trees on insertion/deletion to guarantee that they are balanced

Options: red-black trees, AVL trees, ...

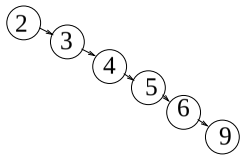
If you are interested, take CS 410.

Balanced Binary Search Tree

To get fast search, we would like search tree to be 'bushy' rather than 'skinny'.



Balanced Tree



Unbalanced Tree

Balanced tree: for every node,
height of left subtree = height of right subtree +/- 1

Hash Tables

Compromise between arrays and recursive data structures

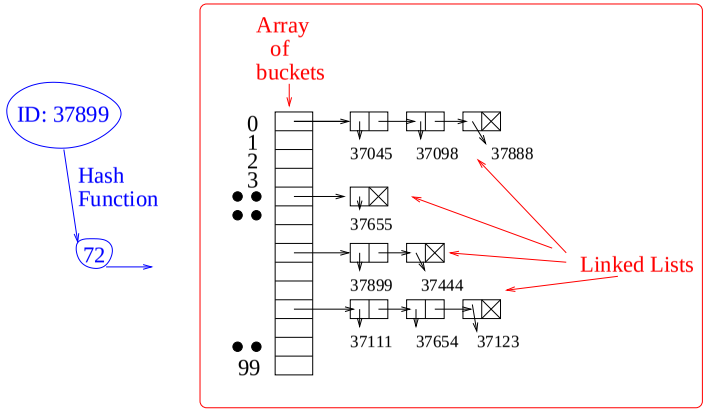
Problem with arrays: they do not grow dynamically

Recursive data structures:
- advantage: grow on demand as elements are inserted into data structure
- advantage: no need to preallocate worst-case amount of storage
- disadvantage: relatively complicated code for maintaining data structures with good performance (such as balanced binary trees)

A pragmatic compromise: hash tables

Let us design a hash table to permit fast lookup of student IDs (int's) in class.

Hash Tables



Hash Table with 100 Buckets

Performance of Hash Tables

Affected by many factors:

- Size of hash table relative to number of entries
  Consider limit where there is only 1 bucket
  => as bad as simple linked lists!
- Quality of hash function
  Good hashing functions do not lead to 'clustering' of entries
  Bad hashing functions for IDs
  1. constant functions: Hash(ID) = 7
  2. Two most significant digits: Hash(379988) = 37

  Good hashing functions for IDs:
  1. Two least significant digits: Hash(379988) = 88
  2. Sum of digits pairs mod 100: Hash(379988) = 37+99+88 = 224
     => 24

One popular hashing function: square number and take middle digits

Algorithms:

Hash function: function that converts a student ID into a bucket number (integer between 00 and 99 for our example)

Insertion:

1. Hash student ID to get bucket number
2. Append student ID to list at that bucket

Search:

1. Hash student ID to get bucket number
2. Look for ID by walking down list at that bucket

Deletion:

1. Hash student ID to get bucket number
2. Walk down list at that bucket and remove ID from that list.

So far, we have stored only integers into hash tables. In general, we want to store objects.

Two step process:

- Let each object have a hash code which is an integer that corresponds to that object. Java method: *hashCode()*.
  Contract for *int hashCode();* method:
  - Whenever it is invoked in the same object, it must return the same result.
  - Two objects that *equal* must have the same hash codes.
  - Two objects that are not equal should return different hash codes, but are not required to do so.
  - Examples: for Integer objects, hashCode() returns the int contained by the object; for Float objects, it returns bit representation of the floating point number.
- To store/retrieve an object, first extract its hash code, and then use hash code to determine bucket number.

Hashtables in Java

- Classes for hash tables: *HashSet, HashMap*
- You can specify an initial size for hash table. When hash table becomes 75% full, it is automatically expanded.
- Instance method in class Object: *int hashCode();*
- When you define your own class, you can override the hashCode method to respect your class's notion of *equal*.

Hash tables are popular in practice because code is easy to write and maintain, and performance of data structure is good.

Complexity of insertion/deletion/lookup: analysis is quite complex

Our version of hash table is called hash table with separate lists or chained hashing.

Other versions of hash tables such as open-addressed hash tables are in the literature.