Sequence Structures

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

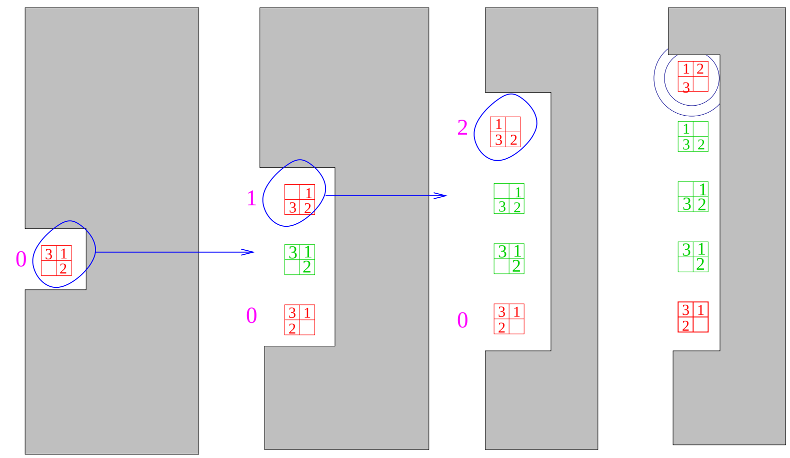
In many applications, we have some domain-specific information that we can exploit to “guide” the graph search.

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

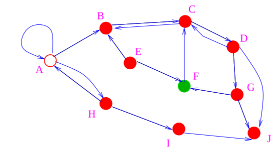
Sometimes, you may not have heuristics to guide your graph search.

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

Two most popular strategies:

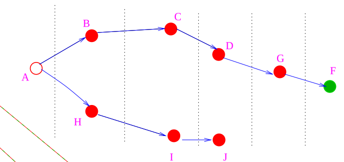
- breadth-first graph search
- depth-first graph search

Breadth-first Search (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

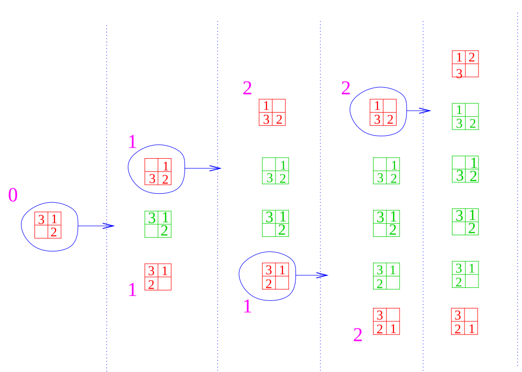
✓ A,B,H,C,I,D,J,G,F

✓ A,H,B,I,C,D,J,G,F

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

Breadth-first search

Sequence structure returns configuration from oldest live generation.



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

How do we implement sequence structure for BFS?

One approach: use priority queue

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

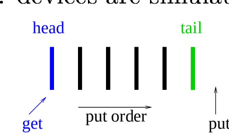
Another approach: use a sequence structure that obeys

First-in-First-out discipline

Queue: get returns item that was put earliest.

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



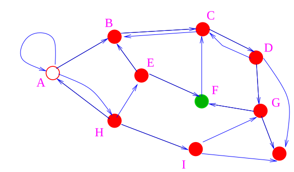
In the context of queues,

put is called enqueue

get is called dequeue

Queue: First-in-first-out Sequence Structure

A,H,B,I,C,D,J,G,F



head tail

Sequence structure: queue [q w e r t y]

[A] → [H B] → [B I E] → [I E C] → [E C J G] → [C J G]

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.

Why use a BFS?

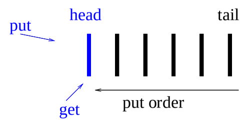
It is easy to show that BFS can determine the shortest path (smallest number of moves) from scrambled configuration to sorted configuration.

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?

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)

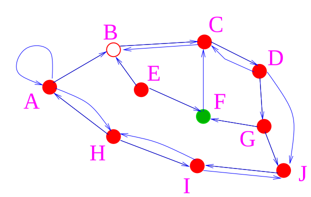


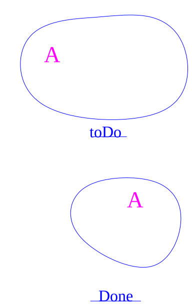
In the context of stacks,

put is known as push

get is known as pop

Example:

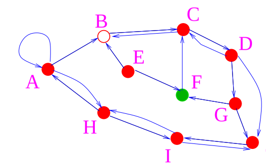




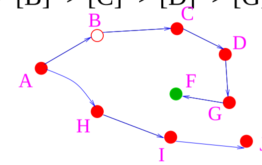
A stack is a LIFO sequence structure.

This graph search strategy is called depth-first graph search.

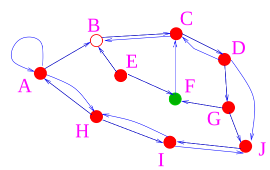
A dfs order:



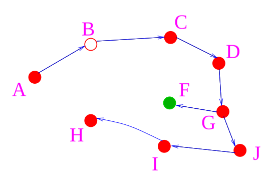
[A] -> [H B] -> [I B] -> [J B] -> [B] -> [C] -> [D] -> [G] -> [F] -> []



Another dfs order:



[A] -> [B H] -> [C H] -> [D H] -> [G J H] -> [F J J H] -> [J J H] -> [I J H] -> [H J H] -> [J H] -> [H] -> []

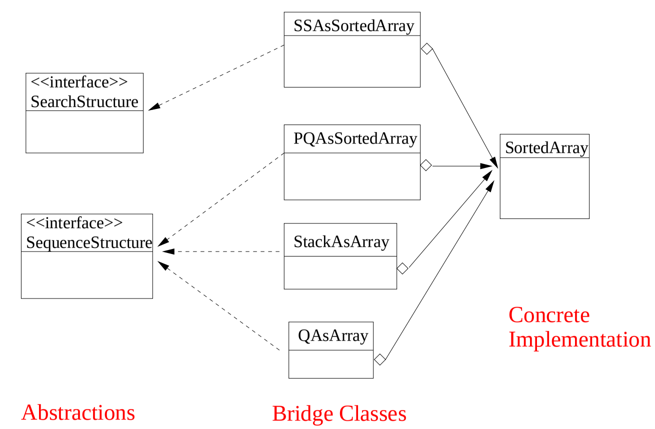


Implementing Sequence Structures Using Sorted Arrays

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

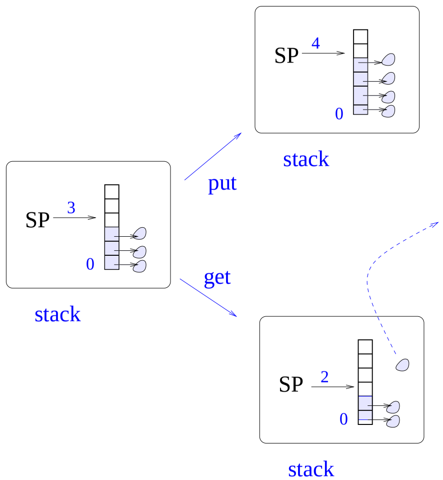
However, we will start with arrays to get a feel for search and sequence structure.

Bridge classes: decoupling abstractions from implementations



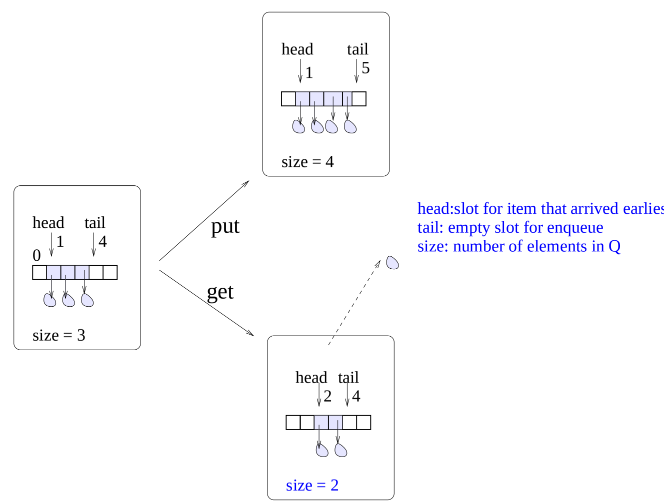
Using Arrays to implement Sequence Structures

Using arrays to implement stacks



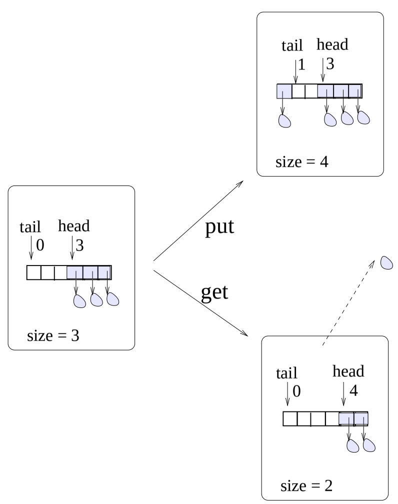
See class StackAsArray at end of handout.

Implementing Queues using arrays



Use the array as a "circular buffer".

Wrap-around: array is a ring buffer!

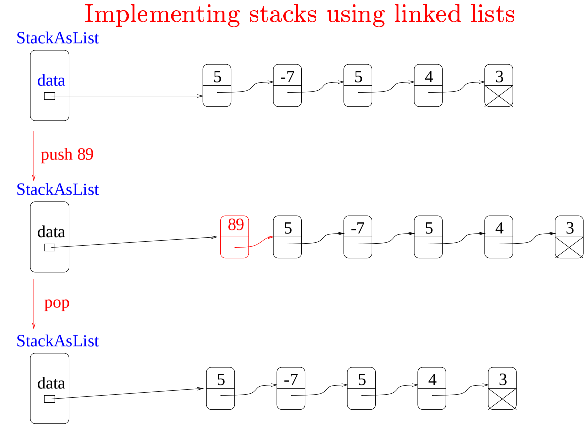


Exercise: can you compute size from values of head and tail?

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

See code in class QAsArray.

Implementing stacks using linked lists



See class StackAsList for an implementation.

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

Implementing queues using linked lists

Head of list: earliest entry

Tail of list: last entry

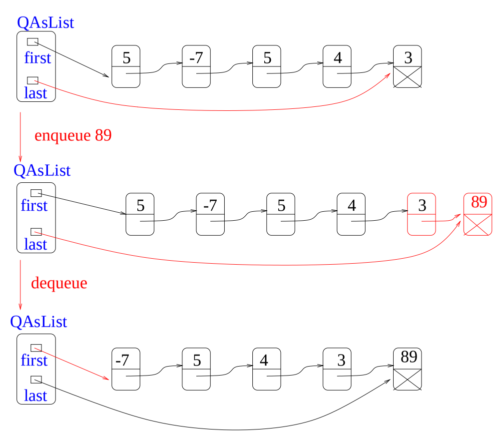
Complication: put and get work at opposite ends of the list.

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.



29

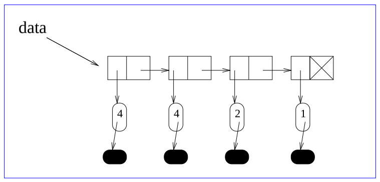
Designing data structures for PriorityQ's

Keep PQ items as sorted list in decreasing order of priority

Entries with same priority are in FIFO order.

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

get: extract from head of list



Code: look at SortedList class discussed earlier

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

30

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.

31

Best priority queue implementation: heap

put: O(log(n)) time

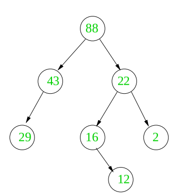
get: O(log(n)) time

32

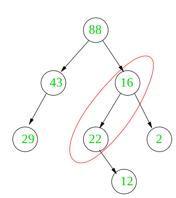
Heap

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



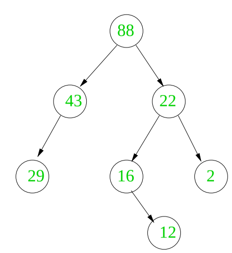
Not a heap

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

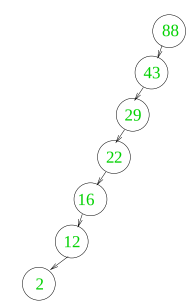
Examples of heaps: ages of people in family tree
Parent is always older than children, but you can have an uncle who is younger than you.

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

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



Heap

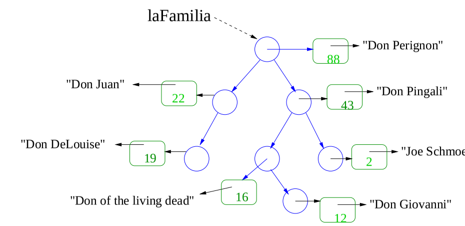


Skinny heap

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

Can we implement a priority queue using a heap?

Question: how do we get and put?

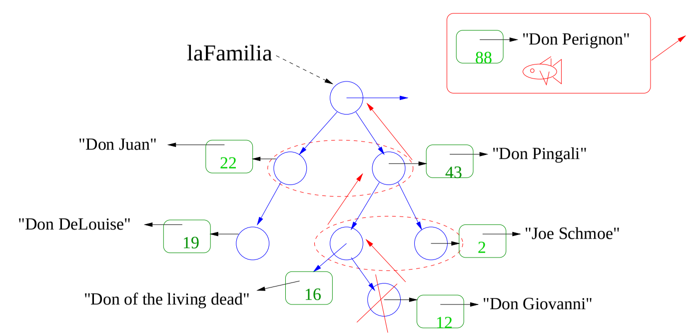
Get: extract the element with largest priority

Put: insert element into data structure and preserve heap property

Let us look at **get** first.

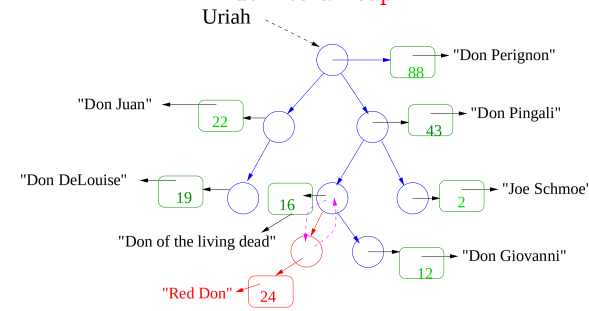
Element to extract is in root of tree (why?).
Removing that element leaves a "hole" in the root.
How should we fill that hole?
One solution:

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.



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.

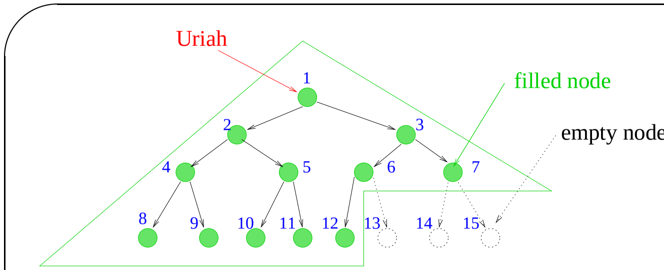
Advantage of heap

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.

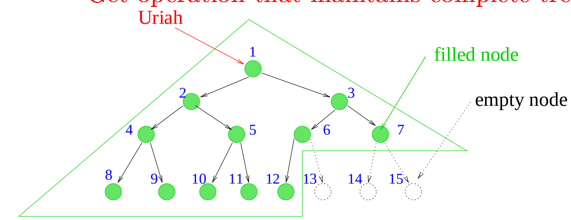
More advanced design to ensure tree is fat and short: ensure heap is a complete binary tree



- "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



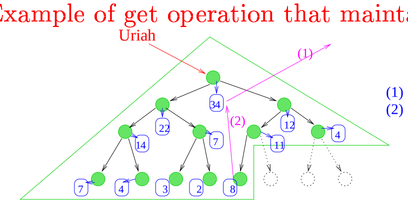
- get algorithm that ensures complete tree-ness:
  - extract element from root of tree and return it
  - in old algorithm for get, we would promote max of nodes 2 and 3 to root, and keep going recursively down the tree. However, this may create a "hole" in some position like 8 or 9 ultimately, and we 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

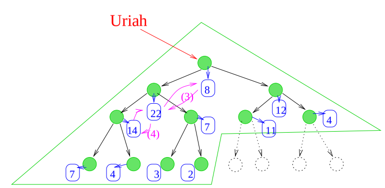
- Use a heap that is a complete binary tree to store PQ elements. Keep track of size of PQ.
- Get: return the root element. Promote the “last” element of the complete binary tree to the root position and walk down the tree restoring heap property. Accessing last position: use binary representation of 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.

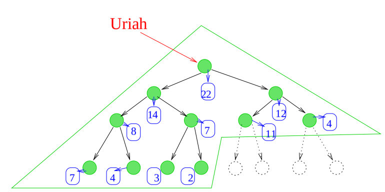
Example of get operation that maintains complete tree-ness



(1) extract root
(2) promote "last" element into root



(3) exchange 8 and 22
(4) exchange 8 and 14



final heap: complete tree once again