Priority queues implemented via heaps

Comp Sci 1575 Data Structures

Missouri S&T | Computer Science

Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.
- Rob Pike

https://en.wikipedia.org/wiki/KISS_principle

Outline

1. Introduction
    - Goal
    - Structure
    - Partial ordering
2. Implementation
    - Array based binary tree
        - Indexing
3. Functions
    - Enqueue
    - Dequeue
    - Build heap
        - Repeated insertion
        - Repeated sift down
    - Other supporting functions
    - Complexity
4. std:: heap and priority queue
    - Algorithms

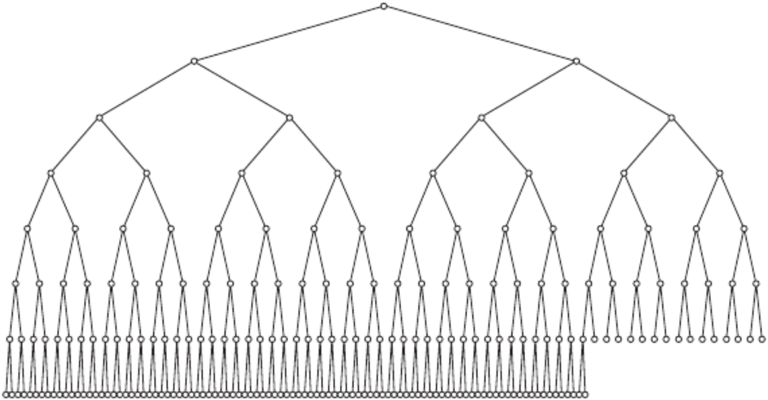
Outline

Priority queue: most important first

- Recall: queue is FIFO
- A normal queue data structure would not implement a priority queue efficiently because search for the element with highest priority would take $\Theta(n)$ time.
- A list, whether sorted or not, would also require $\Theta(n)$ time for either insertion or removal.
- A BST that organizes records by priority could be used to find an item in $\Theta(\log n)$, and the same for insert and remove.
- How could we design and sort a tree so that the highest priority items are most quickly accessible?

Outline

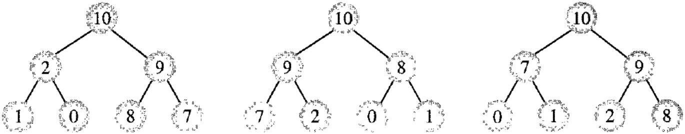
Heaps must be complete trees



By comparison, any given BST can be complete, but a heap is required to be (at insertion, deletion, and construction)

Outline

Heaps are sorted top to bottom

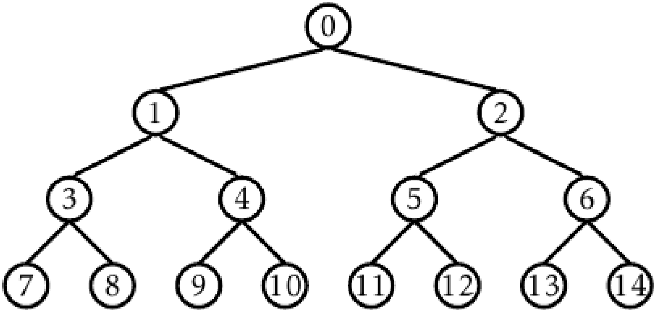


- Value in a node $\geq$ the values in the node's children (Max heap)
- Alternatively, min heap has minimum at top root
- BST is full ordering (left to right, and thus top to bottom also), while heap is partial ordering (just top to bottom, but not left/right); no sibling relationships specified, so left or right child can be the larger of the two children, but both must be smaller than the parent.
- Not as good as BST for finding an arbitrary value in the collection; heap would be O(n) for that, but better for finding most extreme value; merely O(1)
- Like "normal" queue, we are not interested in finding an arbitrary value.

Outline

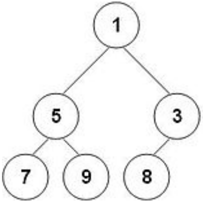
Outline

Trees as arrays



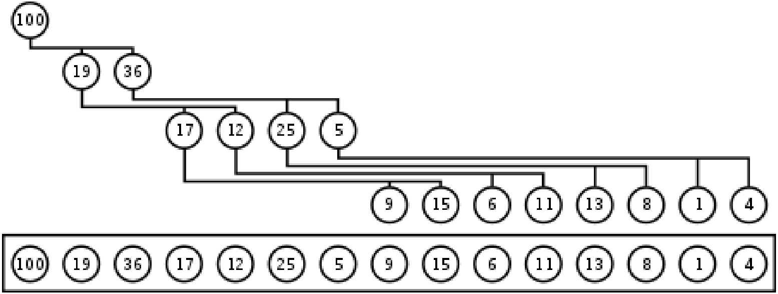
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Trees as arrays



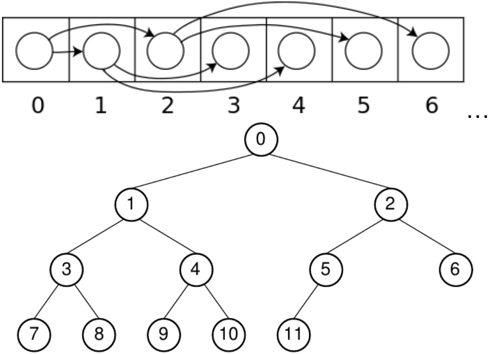
Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Trees as arrays



Array based binary tree

The total number of nodes in the tree is n. The index of the node in question is r, which must fall in the range 0 to n - 1.



- Parent$(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
- Left child$(r) = 2r+1$ if $2r+1 < n$.
- Right child$(r) = 2r+2$ if $2r+2 < n$.
- Left sibling$(r) = r-1$ if r is even.
- Right sibling$(r) = r+1$ if r is odd and $r+1 < n$.

How fast are indices?

Indexing in heaps

Which indices are important for a heap?
Which nodes elements need to be compared?

Indexing schemes

Let n be the number of elements in the heap and i be an arbitrary valid index of the array storing the heap.

If the tree root is at index 0, with valid indices 0 through $n-1$, then each element a at index i has

- children at indices $2i+1$ and $2i+2$
- its parent at index $floor((i-1)/2)$.

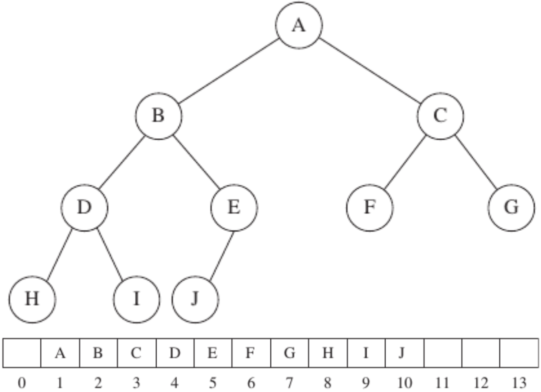
If the tree root is at index 1, with valid indices 1 through n, then each element a at index i has

- children at indices $2i$ and $2i+1$
- its parent at index $floor(i/2)$.

Indexing in heaps

If you scoot the first element back to 1,
then for any element in array position i:

- the left child is in position $2i$,
- the right child is in the cell after the left child $(2i + 1)$,
- the parent is in position $\lfloor i/2 \rfloor$



Counter to this picture, in this class we start at 0

Formal definition of a heap

Heap array formal definition: Node $\geq$ than its children

- With array starting at 0, and i being each node:
  $heap[i] \geq heap[2 * i + 1]$, for $0 \geq i \geq \frac{n-1}{2}$
  $heap[i] \geq heap[2 * i + 2]$, for $0 \geq i \geq \frac{n-2}{2}$

Outline

Priority queue as heap

What are the main operations we need in this queue?

Outline

Enqueue

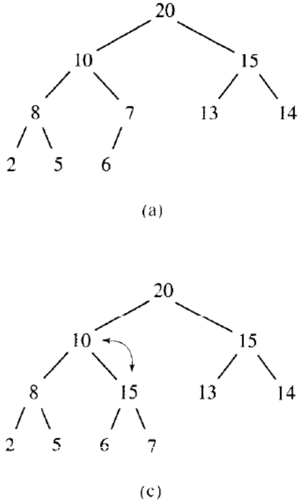
What should enqueue do?
What steps does is require?
Draw it out.

Enqueue: high level design first

1. Place the new entry in the heap in the first available location. This keeps the structure as a complete binary tree. However, it might no longer be a heap, since the new entry might have a higher value than its parent
2. while (new entry has priority that is higher than its parent) swap the new entry with its parent

Enqueue 15 in max heap

Inserting 15 in max heap:



(a) Put 15 at end; (b) Swap 15 with 7
(c) Swap 15 with 10; (d) Done!

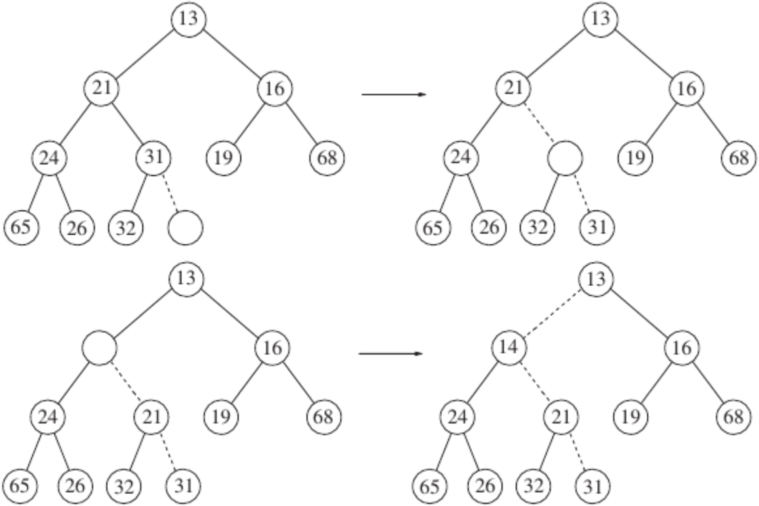
Enqueue and sift up

```
Place x in heap in first available location
(to maintain a complete binary tree).
while (x > parent)
  Swap x with its parent
  Stop when x becomes root or
  when parent is no longer < x
```

```
heapEnqueue(e1)
  put e1 at the end of heap;
  while e1 is not the root and e1 > parent(e1)
    swap e1 with its parent
```

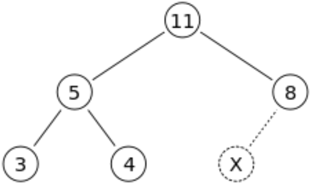
Enqueue and sift up (Note: min-heap in image)

Enqueue 14 in min heap:

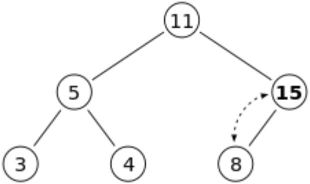


- Walk through steps

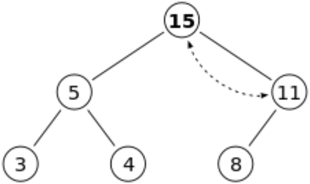
Engueue 15 in max heap



1 add element



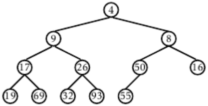
2 swap 15 with 8



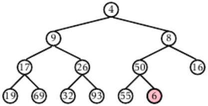
3 swap 15 with 11

What is Θ for this function? Is tree always balanced?

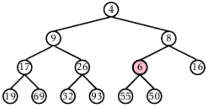
Enqueue 6 in min heap



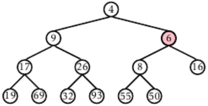
4	9	8	17	26	50	16	19	69	32	93	55			



4	9	8	17	26	50	16	19	69	32	93	55	6		



4	9	8	17	26	6	16	19	69	32	93	55	50		



4	9	6	17	26	8	16	19	69	32	93	55	50		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Outline

Dequeue

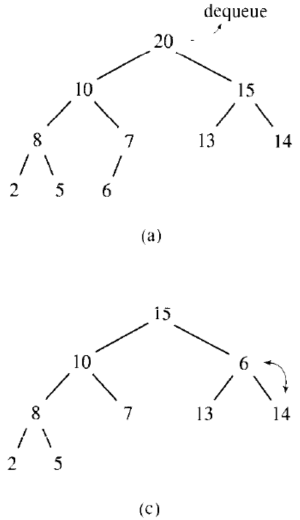
What should dequeue do?
What steps does it require?
Draw it out

Dequeue: high level design first

1. Copy the entry at the root of the heap to the variable that is used to return a value
2. Copy the last entry in the deepest level to the root, and take that last node out of the tree. This entry is now "out of place"
3. while(the "out of place" entry has a priority that is lower than any of its children)
   swap the "out of place" entry with its highest priority child

Why highest priority child?

Dequeue root in max heap (happens to be 20)



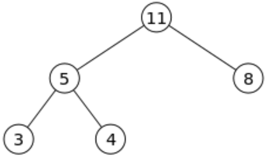
(a) Remove root (20) and move end (6) to root;
(b) Swap 6 with higher priority child (15);
(c) Move end (6) to root ; (d) Done!

Dequeue and sift down

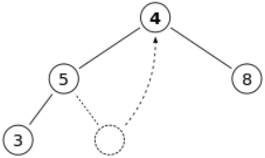

```
Remove rightmost deepest entry; call it x.
Make x the new root.
while (x < one of its children))
  swap x with its largest child
  Stop when x becomes a leaf or
  when x is no longer < one of its children
```

```
heapDequeue()
  extract the element from the root;
  move element from last leaf to its place;
  remove the last leaf;
  p = the root;
  while p is not a leaf and p < its children
    swap p with the larger child;
```

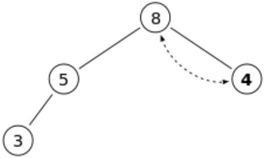
Dequeue root (11) in max heap



1 Copy root



2 Overwrite root with last

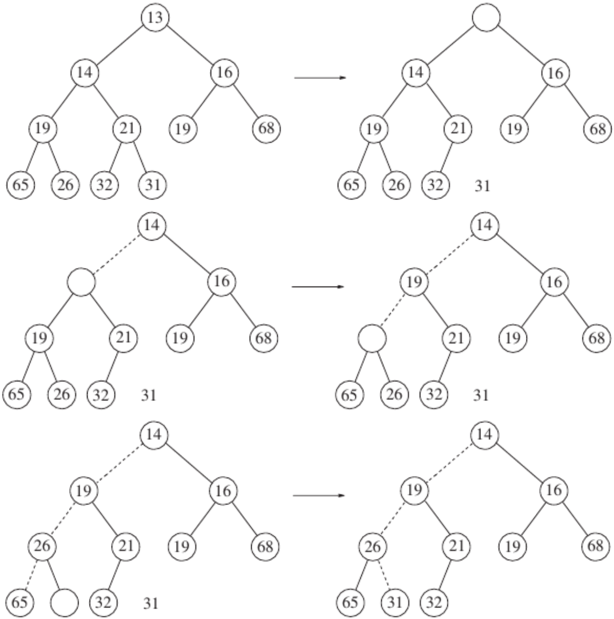


3 swap down

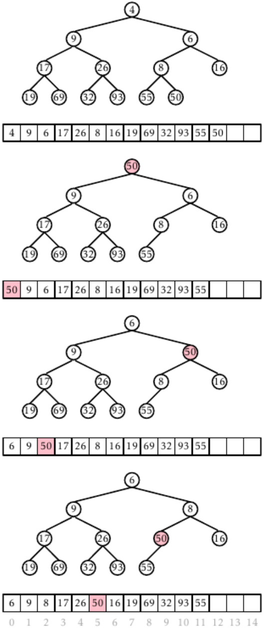
What is Θ for this function?

Why choose one of the "worst" nodes to replace root?

Dequeue (13) in min heap



Dequeue root of min heapi (4)

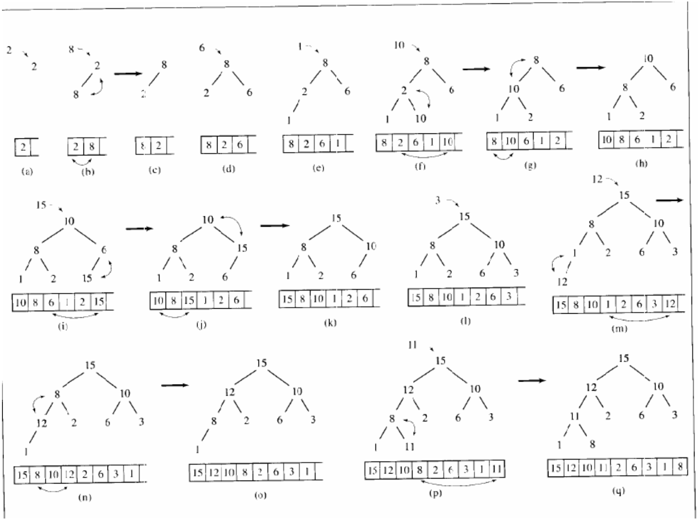


Outline

Building heaps

How to build a heap from a randomly sorted complete tree (which is a randomly sorted array, if we assume an array-tree data structure)?

Heapify via repeated insertion



- $\Theta(n \log n)$

A helper method for efficient heapify: Sift down

Sift down was already part of Dequeue

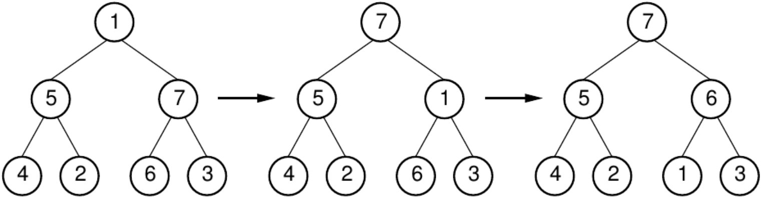
```
siftDown(A, i):
  left = 2*i
  right = 2*i + 1
  largest = i

  if left <= heap_length(A) and A[left] > A[largest]:
    largest = left
  if right <= heap_length(A) and A[right] > A[largest]:
    largest = right

  if largest != i then:
    swap A[i] and A[largest]
    siftDown(A, largest)
```

For the above algorithm to re-heapify the array, the node at index i and at least one of its direct children must violate the heap property. If they do not, the algorithm will fall through with no change to the array.

Sift down: modular method

Sift 1 down:

- Swap 1 (element 0) with 7 (larger child)
  Why higher priority child?
- Swap 1 (element 2) with 6 (larger child)
  Would it be a heap if we promoted lower priority child?



What about reorganizing the whole array?

- Can we repeatedly apply sift down to randomly sorted complete binary tree to form a heap?
- How do we cover all nodes?
- Where do we start?

Build via repeated sift-down


```
buildMaxHeap(A):
  // heap is set to same size as array
  heap_length[A] = length[A]

  // going backwards
  for each index i from floor(length[A]/2) to 1 do:
    siftDown(A, i)
```

Which node does this start with?
Which node does it end with?
What is Θ for this function?

Outline

Other supporting functions

Some housekeeping functions are also helpful (see heap.cpp)

Outline

Complexity

Depending on tree type:

Algorithm	Average	Worst Case
Find-min	O(1)	O(1)
Insert	O(2.607)-why?	O(log n)
Space	O(n)	O(n)
Search	O(n)	O(n)
Delete	O(log n)	O(log n)
Peek	O(1)	O(1)
Build via repeated insert	O(n)	O(n log n)
Build via repeat sift-down	O(n)	O(n)

Operation	Binary[5]	Leftist	Binomial[5]	Fibonacci[5][6]	Pairing[7]	Brodal[8][a]	Rank-pairing[10]	Strict Fibonacci[11]
find-min	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete-min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$
insert	$O(\log n)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$[b]	$o(\log n)$[b][c]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
merge	$\Theta(n)$	$\Theta(\log n)$	$O(\log n)$[d]	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

We just reviewed Binary heap
(where insert is not entirely correct).

Outline

Outline

Algorithms: example of heap functions

Defines functions for a variety of purposes (e.g. searching, sorting, counting, manipulating) that operate on ranges of elements. Range is defined as [first, last) where last refers to the element past the last element to inspect or modify.
`http://en.cppreference.com/w/cpp/algorithm/`
`http://www.cplusplus.com/reference/algorithm/`

For example, heap operations can be performed on a vector:

- **is_heap** checks if the given range is a max heap
- **is_heap_until** finds the largest subrange that is a max heap
- **make_heap** creates a max heap out of a range of elements
- **push_heap** adds last-1 element to a max heap
- **pop_heap** removes the largest element from a max heap by moving to end
- **sort_heap** turns a max heap into a range of elements sorted in ascending order

Algorithms: example of heap functions

See: Heap_algorithms.cpp

How can we do this more directly?

std::priority_queue

$\#include < queue >$

- Priority queue is a **container adaptor** that provides constant time lookup of the largest (by default) element, at the expense of logarithmic insertion and extraction.
- User-provided "Compare" can be supplied to change the ordering, e.g., using std::greater$< T >$ would cause the smallest element to appear as the top().
- Working with a priority_queue is similar to managing a heap in some random access container, with the benefit of not being able to accidentally invalidate the heap.

std::priority_queue template parameters


```
template<
  class T,
  class Container = std::vector<T>,
  class Compare = std::less<typename Container::value_type>
>
```

- **T** - The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type.
- **Container** - Type of underlying container to store the elements. Container must satisfy requirements of SequenceContainer, and its iterators must satisfy the requirements of RandomAccessIterator. It must provide the following functions with the usual semantics: front(); push_back(); pop_back();
  Standard containers std::vector and std::deque satisfy these requirements.
- **Compare** - type providing a strict weak ordering.

Demo code: priority queue

See: Priority_queue.cpp