9.2 priority queue and heap implementation

Thursday, November 3, 2022

TODAY

- heap implementation
- priority queue implementation
- creating a heap by doing "heapify"

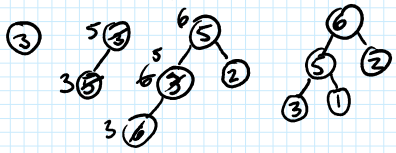
review from last time:

priority queue ADT

templated on priority type P and value V

```
void insert(P priority, V value)  // recall: priorities are not necessarily unique
V remove()
V peek()
P peekPriority()
bool isEmpty()
int getSize()
```

1. Draw the heap representation of this priority queue.
2. pq.peek() returns mango.
   pq.peekPriority() returns 6.



example: most popular fruits in CS35

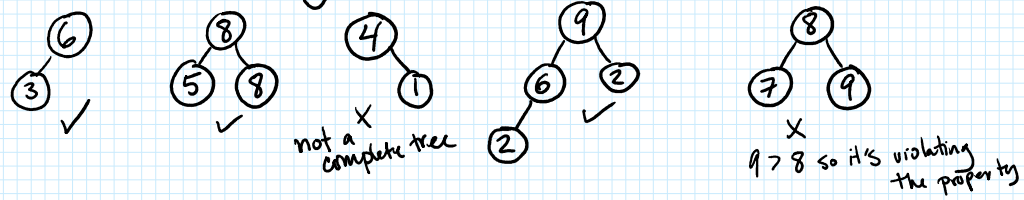
```
HeapPriorityQueue<int, string> pq;
pq.insert(3, "cantaloupe");
pq.insert(5, "peach");
pq.insert(2, "strawberry");
pq.insert(6, "mango");
pq.insert(1, "persimmon");
```

more review:

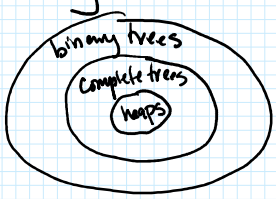
Q: What is a complete tree? binary tree
all levels full except bottom, which is filled left-to-right

Q: What is a heap? a complete binary tree
with the property that, for every node, priority(node) ≥ priority(its descendants)

Q: Which of the following are heaps?



Q: Draw a Venn diagram showing the relationship between:
- binary trees
- heaps
- complete trees



Heap implementation

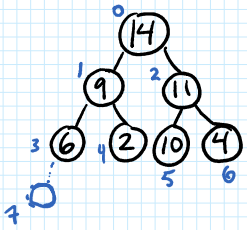
Last time we discussed the algorithms for heap insert and remove.

How will we store and represent heaps in C++?

These operations need to be fast:
- looking up the root
- looking up the parent of a node
- looking up the children of a node
- finding the next "empty" place (to add a node)
- finding the last node (to remove it)

Note: there is only one correct place to look for these spots.

... so we will represent a heap with an array (C++ vector)!



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

Store items in the array in level order.
The heap is a complete tree, so we know there will be no gaps in the array.

Looking up the root? index 0
Looking up the parent of node at index i? parent(i) = $(i-1)/2$ ← integer division so we round down
Looking up the left child of node at index i? left(i) = $2i+1$
Looking up the right child of node at index i? right(i) = $2i+2$

Looking for the next empty spot in the heap? index size
Looking for the last node in the heap? index size-1

How can we check if a particular index has a left or right child?
We can check if left(i) ≥ size to see if the left child exists.
- - - - right(i) ≥ size - - - - right child exists.

IMPLEMENTING PRIORITY QUEUE AS A HEAP

private data:
```
vector<pair<P,V>> heapVector   // vector is C++ arrayList
                               // in each pair, first is priority
                               //               second is value *
```

helper functions:
```
int getLeft(int index)
    return 2*index+1
int getRight(int index)
    return 2*index+2
int getParent(int index)
    return (index-1)/2
```

public methods:

```
V peek() {                                   runtime: O(1)
    if (heapVector.size == 0)
        throw error "You can't peek an empty heap!"
    else
        return heapVector[0].second
}
```

P peekPriority() { runtime: O(1)

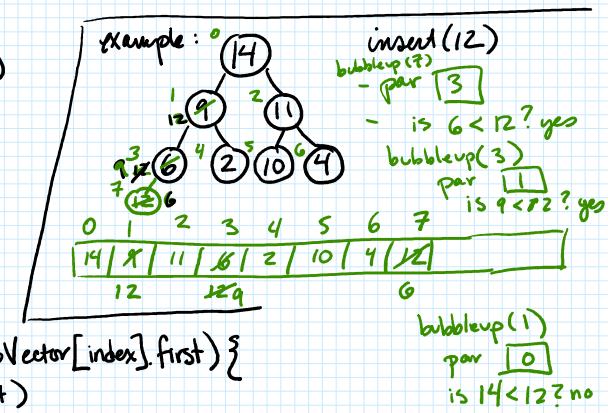
```
}

P peekPriority() {
  if (heapVector.size == 0)
     throw error "You can't peek an empty heap!"
  else
     return heapVector[0].first
}
```

runtime: O(1)

```
void insert(P priority, V value) {
  heapVector.push_back(pair<P,V> (priority, value))
  bubbleUp(heapVector.size -1)
}

void bubbleUp(int index) {
  if (index == 0):
     return
  else {
     int par = getParent(index)
     if (heapVector[par].first < heapVector[index].first) {
        swap(heapVector, index, parent)
        bubbleUp(parent)
     }
  }
}
```

creates a pair and passes it as a parameter

add the parameter given to the next empty spot in the array handles resizing memory and updating size



total work of bubbleUp is $O(\log_2 n)$

Worst case: insert has to resize the array

$O(n) + O(\log_2 n) = O(n)$

Amortized: insert is $O(1) + O(\log_2 n) = O(\log_2 n)$

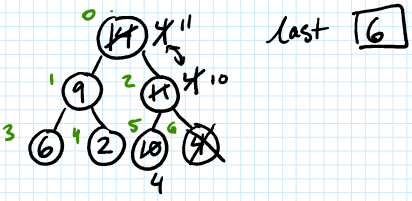
```
V remove() {
  if (heapVector.size == 0)
     throw error "Can't remove from an empty heap!"
  V saved = heapVector[0].second
  int last = heapVector.size -1
  heapVector[0] = heapVector[last]
  heapVector.pop_back() // removes the last item in a vector
  bubbleDown(0)
  return saved
}

void bubbleDown(int index){
  int l = getLeft(index)
  if (l ≥ heapVector.size) // no children
     return
  int r = getRight(index)
  int maxChild
  if (r ≥ heapVector.size) { // only left child
```

example:

remove

save value associated with 14

last 6



bubbleDown(4):

l 11

r 12

maxChild 12

is 4 < 11? yes

Overall runtime:

bubbleDown: is $O(\log_2 n)$

```
int r = getRightChild(index)
int maxChild
if (r >= heapVector.size){ // only left child
    maxChild = l
else {  // has two children
    if (heapVector[l].first >= heapVector[r].first)
        maxChild = l
    else
        maxChild = r
}
// now maxChild is the index of the child with max priority
if (heapVector[index].first < heapVector[maxChild].first) {
    swap(heapVector, index, maxChild)
    bubbleDown(maxChild)
}
}
```

maxChild [2]
is 4 < 11? yes
bubbleDown(2)
l [5]
maxChild [5]
is 4 < 10? yes
bubbleDown(5)
l [11]
no children, done!

bubbleDown:
is $O(\log_2 n)$

remove:
$O(1) + O(\log_2 n)$
$= O(\log_2 n)$

Overall Runtime Comparison For Priority Queues

	sorted array list or sorted linked list	heap
insert	$O(n)$	{amortized $O(\log_2 n)$, worst case $O(n)$}
remove	$O(1)$	$O(\log_2 n)$
peek	$O(1)$	$O(1)$
peekPriority	$O(1)$	$O(1)$

↑ We prefer this implementation.

Using a Heap to Sort

We can use a max heap to get a list of elements in sorted order!

example

We want the output: 16, 11, 7, 5, 4, 3, 2