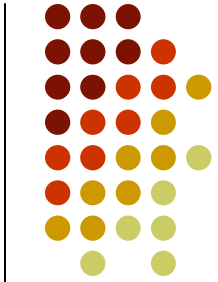
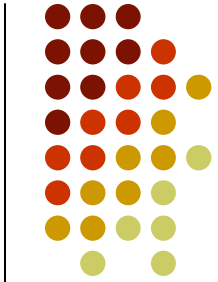
Sorting with Heaps



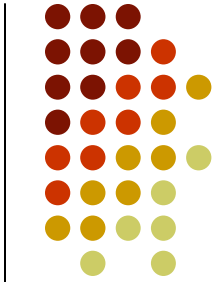
- Observation: Removal of the largest item from a heap can be performed in O(log *n*) time
- Another observation: Nodes are removed in order
- Conclusion: Removing all of the nodes one by one would result in sorted output
- Analysis: Removal of *all* the nodes from a heap is a O(*n**log*n*) operation

But …



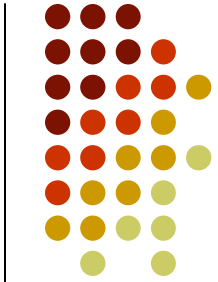
- A heap can be used to return sorted data
  - in O(*n**log *n*) time
- However, we can't assume that the data to be sorted just happens to be in a heap!
  - Aha! But we can put it in a heap.
  - Inserting an item into a heap is a O(log *n*) operation so inserting n items is O(*n**log *n*)
- But we can do better than just repeatedly calling the insertion algorithm

Heapifying Data



- To create a heap from an unordered array repeatedly call bubbleDown
  - bubbleDown ensures that the heap property is preserved from the start node down to the leaves
  - it assumes that the only place where the heap property can be initially violated is the start node; i.e., left and right subtrees of the start node are heaps
- Call bubbleDown on the upper half of the array starting with index *n*/2-1 and working up to index 0 (which will be the root of the heap)
- bubbleDown does not need to be called on the lower half of the array (the leaves)

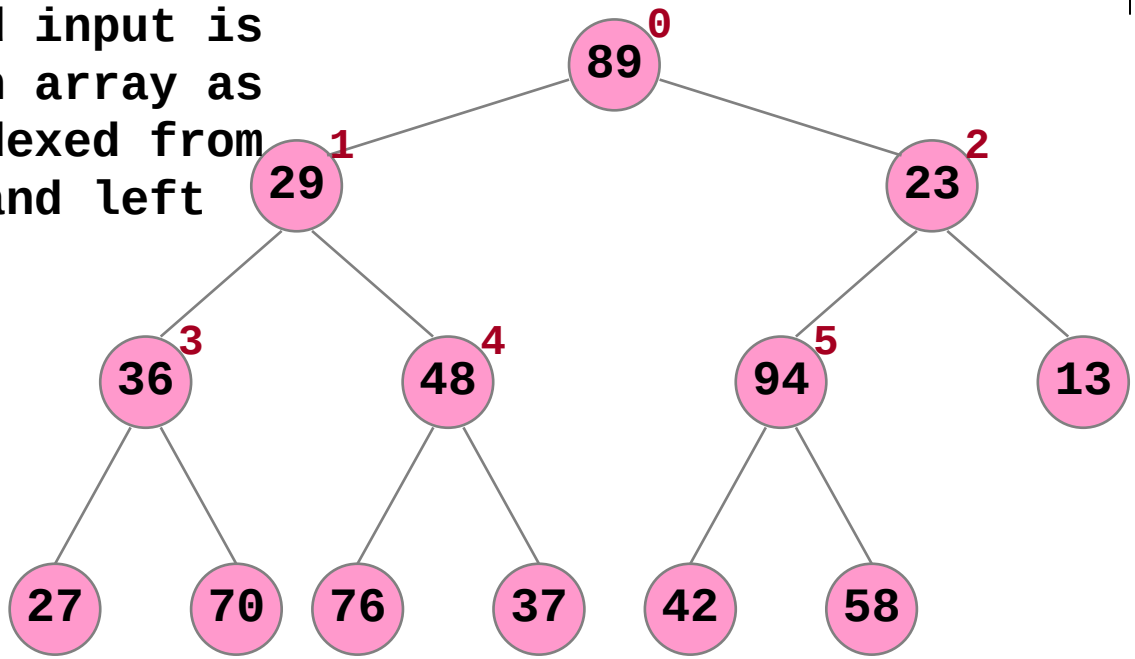
BubbleDown algorithm (part of deletion algorithm)



```
public void bubbleDown(int i)
  // element at position i might not satisfy the heap property
  // but its subtrees do -> fix it
  {
    T item = items[i];
    int current = i; // start at root
    while (left(current) < num_items) { // not a leaf
      // find a bigger child
      int child = left(current);
      if (right(current) < num_items &&
          items[child].getKey() <
          items[right(current)].getKey()) {
        child = right(current);
      }
      if (item.getKey() < items[child].getKey()) {
        items[current] = items[child]; // move its value up
        current = child;
      } else
        break;
    }
    items[current] = item;
  }
```

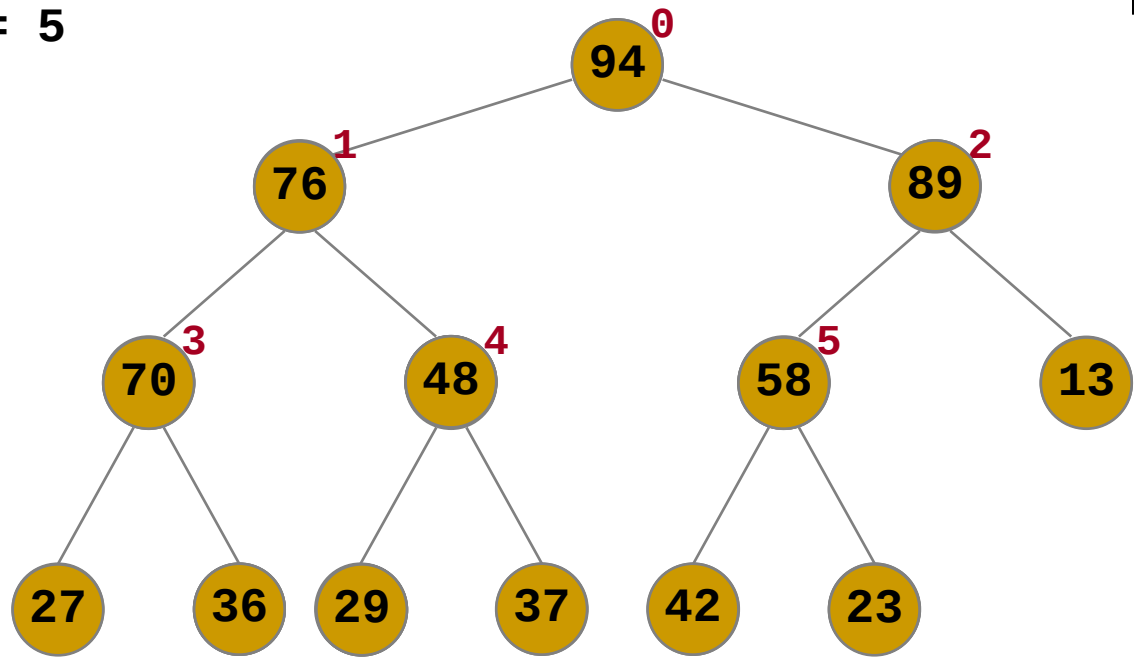
Heapify Example

Assume unsorted input is contained in an array as shown here (indexed from top to bottom and left to right)



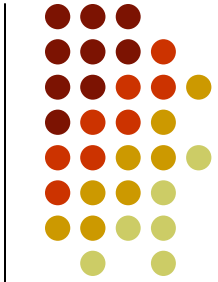
Heapify Example

```
n = 12, n-1/2 = 5
bubbleDown(5)
bubbleDown(4)
bubbleDown(3)
bubbleDown(2)
bubbleDown(1)
bubbleDown(0)
```



note: these changes are made in the underlying array

Heapify algorithm



```
void heapify()
{
  for (int i=num_items/2-1; i>=0; i--)
    bubbleDown(i);
}
```

- Why is it enough to start at position *num_items/2 – 1*?

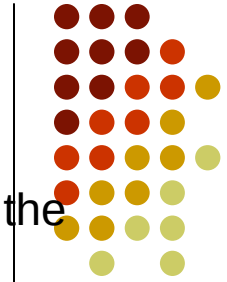
Cost to Heapify an Array

- bubbleDown is called on half the array
  - The cost for bubbleDown is O(*height*)
  - It would appear that heapify cost is O(*n**log*n*)
- In fact the cost is O(*n*)
- The exact analysis is complex (and left for another course)

HeapSort Algorithm Sketch

- Heapify the array
- Repeatedly remove the root
  - At the start of each removal swap the root with the last element in the tree
  - The array is divided into a heap part and a sorted part
- At the end of the sort the array will be sorted (since we have max heap, we put the largest element to the end, etc.)

HeapSort



- assume BubbleDown is static and it takes the array on which it works and the number of elements of the heap as parameters

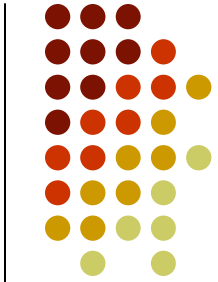
```
public static void bubbleDown(KeyedItem ar[],int
   num_items,int i)
  // element at position i might not satisfy the heap
   property
  // but its subtrees do -> fix it
```

- heap sort:

```
public static void HeapSort(KeyedItem ar[])
  {
    // heapify - build heap out of ar
    int num_items = ar.length;
    for (int i=num_items/2-1; i>-0; i--)
      bubbleDown(ar,num_items,i);

    for (int i=0; i<ar.length-1; i++) {// do it n-1 times
      // extract the largest element from the heap
      swap(ar,0,num_items-1);
      num_items--;
      // fix the heap property
      bubbleDown(ar,num_items,0);
    }
  }
```

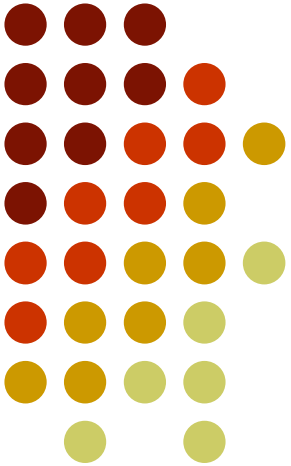
HeapSort Notes



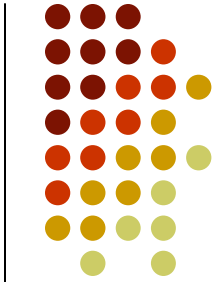
- The algorithm runs in O(*n**log *n*) time
  - Considerably more efficient than selection sort and insertion sort
  - The same (O) efficiency as mergeSort and quickSort
- The sort is carried out “in-place”
  - That is, it does not require that a copy of the array to be made (memory efficient!) – quickSort has a similar property, but not mergeSort

CMPT 225

Hash Tables

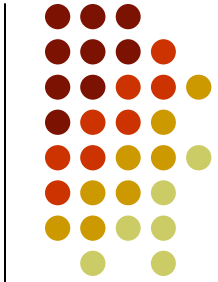


Is balanced BST efficient enough?



- What drives the need for hash tables given the existence of balanced binary search trees?:
  - support relatively fast searches (O (log n)), insertion and deletion
  - support range queries (i.e. return information about a range of records, e.g. “find the ages of all customers whose last name begins with ’S’”)
  - are dynamic (i.e. the number of records to be stored is not fixed)
- But note the “*relatively fast searches*”. What if we want to make many single searches in a large amount of data? If a BST contains 1,000,000 items then each search requires around $\log_2$ 1,000,000 = 20 comparisons.
- If we had stored the data in an array and could (somehow) know the index (based on the value of the key) then each search would take constant (O(1)) time, a twenty-fold improvement.

Using arrays

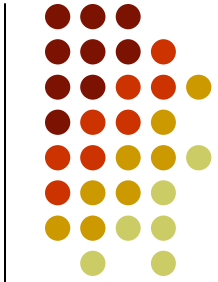


- If the data have conveniently distributed keys that range from 0 to the some value N with no duplicates then we can use an array:
  - An item with key K is stored in the array in the cell with index K.
  - Perfect solution: searching, inserting, deleting in time O(1)
  - Drawback: N is usually huge (sometimes even not bounded) – so it requires a lot of memory.

- Unfortunately this is often the case. Examples: we want to look people up by their phone numbers, or SINs, or names.

- Let’s look at these examples.

Using arrays – Example 1: phone numbers as keys



- For phone numbers we can assume that the values range between 000-000-0000 and 999-999-9999 (in Canada). So let's see how big an array has to be to store all the possible numbers. It's easy to map phone numbers to integers (keys), just get rid of the '-'s. So we have a range from 0 to 9,999,999,999. So we'd need an array of size 10 billion. There are two problems here:
  - The first is that you won't fit the array in main memory. A PC with 2GB of RAM can store only 536,870,912 references (assuming each reference takes only 4 bytes) which is clearly insufficient. Plus we have to store actual data somewhere.

    (We could store the array on the hard drive, but it would require 40GB.)
  - The other problem is that such an array would be horribly wasteful. The population of Canada estimated in July 2004 is 32,507,874, so if we assume that that's the approx. number of phone numbers, there is a huge amount of wasted space.