CS 261 – Data Structures

BuildHeap and Heap Sort

Tim Budd
Ron Metoyer
Sinisa Todorovic
Given an array of data,
construct the heap
Heap Implementation: Constructors

```c
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i =
```
Heap Implementation: Constructors

```c
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
        /* Make the heap from the subtree rooted at i */
        _adjustHeap(da, maxIdx, i);
}
```
Heap Implementation: Constructors

```c
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
        /* Make the heap from the subtree rooted at i */
        _adjustHeap(da, maxIdx, i);
}
```
At the beginning, only the leaves are proper heaps:
Leaves are all nodes with indices > maxIdx / 2

Why?

At the beginning, only the leaves are proper heaps:
Leaves are all nodes with indices > maxIdx / 2
Heap Implementation: **Constructors**

```c
void buildHeap(struct dynArray * da) {
    int maxIdx = da->size;
    int i;
    for (i = maxIdx / 2 - 1; i >= 0; i--)
        /* Make the heap from the subtree rooted at i */
        _adjustHeap(da, maxIdx, i);
}
```

At each step, the subtree rooted at \( i \) becomes a heap.
Heap Implementation: Build heap

```c
void buildHeap(struct dynArray * da) {
    int maxIdx = ds->size;
    int i;
    for (i = max / 2 - 1; i >= 0; i--)
        _adjustHeap(da, maxIdx, i); /* Make subtree rooted at i a heap */
}
```

- For all subtrees that are not already heaps:
  - Call `_adjustHeap` with the `largest` node index that is not already guaranteed to be a heap
  - Iterate until the root node becomes a heap
Heap Implementation: **Build heap**

```c
void buildHeap(struct dynArray * da) {
    int maxIdx = ds->size;
    int i;
    for (i = max / 2 - 1; i >= 0; i--)
        _adjustHeap(da, maxIdx, i); /* Make subtree rooted at i a heap */
}
```

- Why call `_adjustHeap` with the *largest* node index?
  
  - Because its children, having larger indices, are already guaranteed to be heaps
Heap Implementation: \_\texttt{adjustHeap}

First iteration: adjust largest non-leaf node (index 4)

\[i = \left(\frac{\text{max}}{2} - 1\right)\]
Heap Implementation: adjustHeap (cont.)

Second iteration: adjust largest non-heap node (index 3)
Heap Implementation: `_adjustHeap` (cont.)

Third iteration: adjust largest non-heap node (index 2)
Heap Implementation: **adjustHeap** (cont.)

Fourth iteration: **adjust**

largest non-heap node (index 1)

Already heaps
Heap Implementation: \texttt{adjustHeap} (cont.)

Fifth iteration: adjust largest non-heap node (index 0 $\rightarrow$ root)

Already heaps

\texttt{adjustHeap}
Heap Implementation: adjustHeap (cont.)

Already heaps (entire tree)
Heap Implementation: **Sort Descending**

Sorts the data in descending order:

1. Builds heap from initial (unsorted) data
2. Iteratively swaps the smallest element (at index 0) with last *unsorted* element
3. Adjust the heap after each swap, but only considers the *unsorted* data
Heap Implementation: Sort Descending

```c
void heapSort(struct dyArray * data) {
    int i;

    buildHeap(data);  /* Build initial heap*/
    for (i = sizeDynArr(data)-1; i > 0; i--){
        /*Swap last element with the first*/
        TYPE tmp = getDynArr(data,i);
        putDynArr(data,i,getDynArr(data,0));
        putDynArr(data,0,temp);  /*Sorts in reverse order*/
        _adjustHeap(data,i,0);  /*Rebuild heap property*/
    }
}
```
Heap Analysis: **Sort**

- Execution time:
  - Build heap:
    - $n$ calls to `adjustHeap` = $n \log n$
  - Loop:
    - $n$ calls to `adjustHeap` = $n \log n$
  - Total:
    - $2n \log n = O(n \log n)$
Heap Analysis: **Sort**

- Advantages/disadvantages:
  - Same average as merge sort and quick sort
  - Doesn’t require extra space as the merge sort does
  - Doesn’t suffer if data is already sorted or mostly sorted
Practice: Breadth-First Traversal

```c
void PrintBreadthFirstBST(struct BSTree *tree){
    struct listQueue *q;
    struct BSTNode *current = tree->root;
    initListQueue(q);
    addBackListQueue (q, current);
    ...
}
```

Getting from the front

Adding to the back
void PrintBreadthFirstBST(struct BSTree *tree){
    struct listQueue *q;
    struct BSTNode *current = tree->root;
    initListQueue(q);
    addBackListQueue(q, current);
    while(!isEmptyListQueue(q)){
        current = getFrontListQueue(q);
    }
}
void PrintBreadthFirstBST(struct BSTree *tree){
    struct listQueue *q;
    struct BSTNode *current = tree->root;
    initListQueue(q);
    addBackListQueue (q, current);
    while(!isEmptyListQueue(q)){
        current = getFrontListQueue(q);
        removeFrontListQueue(q);
        printf("%f ",current->val);
    }
}
```c
void PrintBreadthFirstBST(struct BSTree *tree){
    struct listQueue *q;
    struct BSTNode *current = tree->root;
    initListQueue(q);
    addBackListQueue (q, current);
    while(!isEmptyListQueue(q)){
        current = getFrontListQueue(q);
        removeFrontListQueue(q);
        printf("%f  ",current->val);
        if(current->left != 0)
            addBackListQueue (q, current->left);
        if(current->right != 0)
            addBackListQueue (q, current->right);
    }
}
```