Heaps / Priority Queues

Computing Laboratory

http://www.isical.ac.in/~dfslab

Definitions

Complete binary tree

- All non-leaf nodes have exactly two children.
- All leaf nodes are at the same depth in the tree.

$\Rightarrow$ Complete binary tree of height h contains $2^{h+1} - 1$ nodes.

Full binary tree

If the height of the tree is h, then

- leaf nodes can occur only at depth h and $h - 1$
- at most one non-leaf node can have one children
- at level $h - 1$:
  - all nodes with 2 children occur to the left of nodes with < 2 children
  - any node with 1 child occurs to left of nodes with no children

Definitions

Max-heap property

Key value of any node $\geq$ key value of each of its children

Max heap

Full binary tree that satisfies the max-heap property

Priority queues

Reference: Sedgewick and Wayne, Section 2.4

- ADT supporting the operations INSERT, DELETE-MAX (or DELETE-MIN)
- May be implemented using
  - arrays / linked lists (sorted / unsorted)
  - heaps

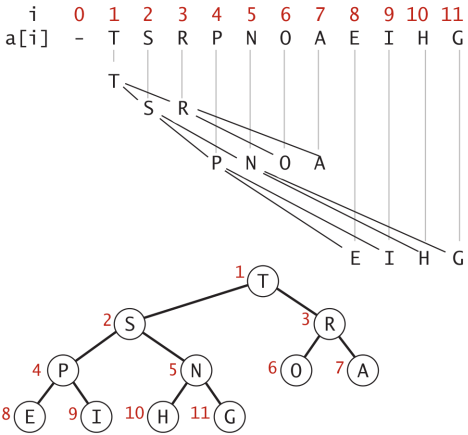
data structure	insert	remove maximum
ordered array	N	1
unordered array	1	N
heap	$\log N$	$\log N$

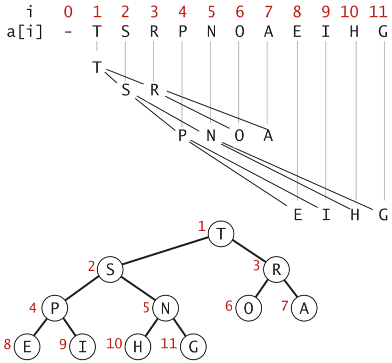
Table from `http://algs4.cs.princeton.edu/24pq/`

Implementation

Array representation

Full binary trees may be represented by an array, with nodes stored in *level order*.





One-based indexing (SEDGEWICK AND WAYNE)

- Children of node at i are at $2i$ and $2i + 1$
- Parent of node at i is at $\lfloor i/2 \rfloor$

Zero-based indexing

- Children of node at i are at $2i + 1$ and $2i + 2$
- Parent of node at i is at $\lfloor (i - 1)/2 \rfloor$

Implementation

```
typedef struct {
    unsigned int num_allocated, num_used;
    int *array; /* one-based indexing used (cf. SEDGEWICK AND WAYNE) */
} INT_HEAP;

void initHeap(INT_HEAP *h) {
    h->num_allocated = INIT_HEAP_SIZE;
    h->num_used = 0;
    if (NULL == (h->array = malloc(h->num_allocated * sizeof(int)))) {
        perror("initHeap: out of memory");
        exit(-1);
    }
    return;
}
```

Insert routine

```
void insert(INT_HEAP *h, int x)
{
    /* First, make sure there's space for another element */
    if (h->num_used + 1 == h->num_allocated) {
        h->num_allocated *= 2;
        if (NULL == (h->array = realloc(h->array, h->num_allocated *
    sizeof(int)))) {
            perror("insert: out of memory");
            exit(-1);
        }
    }
    /* Insert element at end */
    h->num_used++;
    h->array[h->num_used] = x;

    /* Restore heap property */
    swapUp(h, h->num_used);
    return;
}
```

Delete maximum

```c
int deleteMax(INT_HEAP *h)
{
    int max;
    /* Max is at the root (index 1) */
    max = h->array[1];
    /* Copy last element to root */
    h->array[1] = h->array[h->num_used];
    h->num_used--;
    /* Restore heap property */
    swapDown(h, 1);
    return max;
}
```

Auxiliary functions

```
static void swapUp(INT_HEAP *h, int k) {
    int tmp;
    while (k > 1 && (h->array[k/2] < h->array[k])) {
        tmp = h->array[k/2], h->array[k/2] = h->array[k], h->array[k] = tmp;
        k = k/2;
    }
    return;
}

static void swapDown(INT_HEAP *h, int k) {
    int tmp;
    while (2*k <= h->num_used) {
        int j = 2*k;
        /* choose child with larger key */
        if (j < h->num_used && (h->array[j] < h->array[j+1]))
            j++;
        if (h->array[k] >= h->array[j]) break;
        tmp = h->array[k], h->array[k] = h->array[j], h->array[j] = tmp;
        k = j;
    }
    return;
}
```

buildheap / heapify

```
void buildheap(INT_HEAP *h)
{
    int k;

    for (k = h->num_used / 2; k >= 1; k--)
        swapDown(h, k);
    return;
}
```

Heapsort

NOTE: Indexing from 1!

```
void heapsort(int *a, int N) {
    int tmp;
    INT_HEAP h;

    h.num_allocated = h.num_used = N;
    h.array = a;
    /* Make heap out of array */
    buildheap(&h);
    /* Sort by successive deleteMax */
    while (h.num_used > 1) {
        tmp = h.array[1],
            h.array[1] = h.array[h.num_used],
            h.array[h.num_used] = tmp; // move max to end
        h.num_used--;
        swapDown(&h, 1);
    }

    return;
}
```

Heapsort example

a[i]

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

Generic Heaps

Implementation

```
typedef struct {
    size_t element_size; /* generic => need to store this */
    unsigned int num_allocated, num_used;
    void *array;          /* one-based indexing used (cf. SEDGEWICK AND WAYNE) */
    int (*comparator)(void *, int, int); /* returns -ve, 0, or +ve, as for qsort
      */
} HEAP;

void initHeap(HEAP *h, size_t element_size, int (*comparator)(void *, int, int))
      {
    h->element_size = element_size;
    h->num_allocated = 10;
    h->num_used = 0;
    if (NULL == (h->array = malloc(h->num_allocated * element_size))) {
        perror("initHeap: out of memory");
        exit(-1);
    }
    h->comparator = comparator;
    return;
}
```

Auxiliary functions

```c
static void swap(HEAP *h, int i, int j)
{
    /* NOTE: One-based indexing is used. h->array[0] is unused and
     * can be used as the temporary location while swapping
     */
    char *ip = (char *) h->array + i * h->element_size;
    char *jp = (char *) h->array + j * h->element_size;
    char *tp = (char *) h->array;
    memcpy((void *) tp, (void *) ip, h->element_size);
    memcpy((void *) ip, (void *) jp, h->element_size);
    memcpy((void *) jp, (void *) tp, h->element_size);
    return;
}
```

See https://stackoverflow.com/questions/1666224/what-is-the-size-of-void

Insert routine I

```c
static void swapUp(HEAP *h, int k)
{
    while (k > 1 && (h->comparator(h->array, k/2, k) < 0)) {
        swap(h, k, k/2);
        k = k/2;
    }
    return;
}
```

Insert routine II

```
void insert(HEAP *h, void *x)
{
    /* First, make sure there's space for another element */
    if (h->num_used + 1 == h->num_allocated) {
        h->num_allocated *= 2;
        if (NULL == (h->array = realloc(h->array, h->num_allocated * h->
    element_size))) {
            perror("insert: out of memory");
            exit(-1);
        }
    }
    /* Insert element at end */
    h->num_used++;
    memcpy((char *) h->array + h->num_used * h->element_size,
           x,
           h->element_size);
    /* Restore heap property */
    swapUp(h, h->num_used);
    return;
}
```

Delete maximum I

```
static void swapDown(HEAP *h, int k)
{
    while (2*k <= h->num_used) {
        int j = 2*k;
        /* choose child with larger key */
        if (j < h->num_used && (h->comparator(h->array, j, j+1) < 0))
            j++;
        if (h->comparator(h->array, k, j) >= 0) break;
        swap(h, k, j);
        k = j;
    }
    return;
}
```

Delete maximum II

```
void deleteMax(HEAP *h, void *max)
{
    /* Max is at the root (index 1) */
    memcpy(max, h->array + h->element_size, h->element_size);
    /* Copy last element to root */
    memcpy(h->array + h->element_size,
           h->array + h->num_used * h->element_size,
           h->element_size);
    h->num_used--;
    /* Restore heap property */
    swapDown(h, 1);
    return;
}
```

Example comparator routine

```c
static int compare_int(void *array, int i1, int i2)
{
    int n1 = *((int *) array + i1);
    int n2 = *((int *) array + i2);
    return n1 - n2;
}
```

Heapsort

NOTE: Indexing from 1!

```
void heapsort(void *a, int N, size_t element_size,
              int (*comparator)(void *, int, int))) {
  int k;
  HEAP h;

  h.element_size = element_size;
  h.num_allocated = h.num_used = N;
  h.array = a;
  h.comparator = comparator;
  /* Make heap out of array */
  for (k = N/2; k >= 1; k--)
      swapDown(&h, k);
  /* Sort by successive deleteMax */
  while (h.num_used > 1) {
      swap(&h, 1, h.num_used); // move max to end
      h.num_used--;
      swapDown(&h, 1);
  }
}
```

Problems I

0. Use the code given above to create your own `libheap.a`. Please report any bugs that you encounter.
1. Write a program that takes k sorted lists of integers / floating point numbers / strings, and merges them into a single sorted list.
   Input file format:

```
1 # number of test cases
4 # test case 1: number of sorted lists
3 10 20 30 # list 1: number of elements, followed by elements in sorted order
2 1 2      # list 2: as abov
5 5 15 25 30 35 # list 3
4 3 9 27 81     # list 4
```

Problems II

2. You are given n ropes of lengths $l_1, l_2, \ldots, l_n$ respectively. The ropes need to be tied together to form one long rope. At a time, you can only tie two ropes together. Suppose that the cost to tie two ropes together is equal to the sum of their lengths. Write a program that takes $l_1, l_2, \ldots, l_n$ as command line arguments, and prints the minimum cost of joining the ropes together.