- Fibonacci heaps

Priority Queues and Heapsort

- data structure to handle ordered keys that may not be in full sorted order
    - add new keys in sorted order
    - process key with largest value
    - closely related to stacks and queues
- only two necessary operations -> add/insert and remove largest
- a priority queue can be used to support sorting -> insert keys then removal will remove items in reverse sorted order
- used for many more advanced algorithms, including graph searching
- generalized operations:
    - construction with n given items
    - insert
    - remove max
    - change an item's priority
    - remove an arbitrary item
    - join two priority queues
- general API

```
bool empty()
void insert(Item)
Item get_max()
```

Elementary Implementations

- can use unordered array or doubly linked-list (prototypical lazy approach to the problem)
- can use ordered array or doubly linked-list (prototypical eager approach to the problem)
- basic implementation:

```
template <class Item>
class PQ {
private:
    Item *pq;
    int N;
public:
    PQ(int maxN) {
        pq = new Item[maxN]; N = 0;
    }
    int empty() const { return N == 0; }
    void insert(Item item) { pq[N++] = item; }
    Item getmax() {
        int max = 0;
        for (int j = 1; j < N; j++)
            if (pq[max] < pq[j]) max = j;
        exch(pq[max], pq[N-1]);
        return pq[--N];
    }
};
```

- performance:

		remove		find	change	
	insert	max	remove	max	priority	join
ordered array	N	1	N	1	N	N
ordered list	N	1	1	1	N	N
unordered array	1	N	1	N	1	N
unordered list	1	N	1	N	1	1
heap	lgN	lgN	lgN	1	lgN	N
binomial queue	lgN	lgN	lgN	lgN	lgN	lgN
best in theory	1	lgN	lgN	1	1	1

Heap Data Structure

- heap is a data structure that guarantees that a key (in an array) is larger than two other keys at specific positions (corresponds to a binary tree)
- properties:
  - a tree is heap-ordered if the key in each node is larger than or equal to the keys in all of the node's children
  - no node in a heap-ordered tree has a key larger than the root
- definition: a heap is a set of nodes with keys arranged in a complete heap-ordered binary tree (represented as an array)
- the ith element is the parent of the 2ith and the (2i + 1)st element (or element in position i/2 is the parent of the element in position i)
- it follows that the element in position i is larger than the elements in positions 2i and 2i + 1
- heaps facilitate the priority queue operations in logarithmic time (except join)
- these operations move along some path from parent to child or back up
- other heap implementations
  - triply-linked heap-ordered complete trees
  - tournaments
  - binomial queues

Algorithms on Heaps

- heapify - general algorithm that makes a modification to a heap and then traverses the heap to ensure a condition holds
- if a child is larger than a parent the child can be swapped with the parent and so on until the condition is met
- bottom-up heapify:

```
template<class Item>
void fix_up(Item a[], int k) {
    while (k > 1 && a[k / 2] < a[k]) {
        exch(a[k], a[k / 2]);
        k = k / 2;
    }
}
```

- top-down heapify:

```
template<class Item>
void fixDown(Item a[], int k, int N) {
    while (2*k <= N) {
        int j = 2*k;
        if (j < N && a[j] < a[j+1]) {
            j++;
        }
        if (!(a[k] < a[j])) {
            break;
        }
        exch(a[k], a[j]); k = j;
    }
}
```

- heap-based priority queue

```
template<class Item>
class PQ {
private:
    Item *pq;
    int N;
public:
    PQ(int maxN) {
        pq = new Item[maxN+1];
        N = 0;
    }
    int empty() const { return N == 0; }
    void insert(Item item) { pq[++N] = item;  fixUp(pq, N); }
    Item get_max() {
        exch(pq[1], pq[N]);
        fixDown(pq, 1, N-1);
        return pq[N--];
    }
};
```

- properties:
    - insert O(lg(n)) for heap-ordered priority queue
    - remove max O(2 * lg(n)) for heap-ordered priority queue
    - change priority, remove, and replace o(2 * lg(n)) comparisons
- sorting with a priority queue
    - insert and then use get_max to remove in reverse sorted order

```
#include "Priority_queue.cxx"
template<class Item>
void pq_sort(Item a[], int l, int r) {
    int k;
    Priority_queue<Item> pq(r-l+1);
    for (k = l; k <= r; k++) pq.insert(a[k]);
    for (k = r; k >= l; k--) a[k] = pq.getmax();
}
```

- using a priority queue as an unordered list and sorting as above corresponds to selection sort, using a priority queue as an ordered list corresponds to insertion sort

Heapsort

- properties:
    - not stable
    - O(1) extra space (see discussion)
    - O(n * lg(n)) time
    - not really adaptive
- maintain a heap using both sort up and sort down as the heap grows and shrinks
- does not maintain the heap via successive insertions -> maintains by going backward through the heap and creating small subheaps -> this facilitates establishing the heap property inductively
- bottom up heap construction takes linear time

```
template <class Item>
void heapsort(Item a[], int l, int r) {
    int k, N = r-l+1;
    Item *pq = a+l-1;
    for (k = N/2; k >= 1; k--) {
        fixDown(pq, k, N);
    }
    while (N > 1) {
        exch(pq[1], pq[N]);
        fixDown(pq, 1, --N);
    }
}
```

- useful because it is guaranteed to sort n elements in place in n * log(n) time (no worst case like quicksort and no use of additional space)
- properties:
    - uses fewer than O(2 * n * lg(n)) comparisons to sort n elements
    - heap-based selection allows the k-th largest of n items to be found in time proportional to n when k is small or close to n, and in time proportional to n * log(n) otherwise
- NOTE: one important application is finding the k largest of n items, especially if k is small
    - build heap and then remove k items (2 * n + 2k * lg(n))
    - build minimum oriented heap of size k and then perform k "replace the minimum" operations
- improvements:
    - can avoid check for position during sortdown using extra bookkeeping but this is only useful when comparisons are costly
    - build array based heap as a heap-ordered ternary tree (reduces tree height at the cost of finding the largest of three children)

Priority-Queue ADT

- more common to want a handle or pointer to an element that is the max rather than removing the max -> used for change priority and remove

```
template<class Item>
class Priority_queue {
private:
    // Implementation-dependent code
public:
    // Implementation-dependent handle definition
    Priority_queue(int);
    int empty() const;
    handle insert(Item);
    Item get_max();
    void change(handle, Item);
    void remove(handle);
    void join(Priority_queue<Item>&);
};
```

- can use arrays, linked lists, etc for various elementary implementations
- doubly linked list implementation:

```
template <class Item>
class Priority_queue {
private:
    struct node {
        Item item; node *prev, *next;
        node(Item v) {
            item = v;
            prev = 0;
            next = 0;
        }
    };
    typedef node *link;
    link head, tail;
public:
    typedef node* handle;
    Priority_queue(int = 0) {
        head = new node(0);
        tail = new node(0);
        head->prev = tail;
        head->next = tail;
        tail->prev = head;
        tail->next = head;
    }
    int empty() const { return head->next->next == head; }
    handle insert(Item v) {
        handle t = new node(v);
        t->next = head->next;
        t->next->prev = t;
        t->prev = head; head->next = t;
        return t;
    }
    Item get_max() {
        Item max;
        link x = head->next;
        for (link t = x; t->next != head; t = t->next) {
            if (x->item < t->item) {
                x = t;
            }
        }
        max = x->item;
        remove(x);
        return max;
    }
    void change(handle x, Item v) { x->key = v; }
    void remove(handle x) {
        x->next->prev = x->prev;
        x->prev->next = x->next;
        delete x;
    }
    void join(Priority_queue<Item>& p) {
        tail->prev->next = p.head->next;
        p.head->next->prev = tail->prev;
        head->prev = p.tail;
        p.tail->next = head;
        delete tail;
        delete p.head;
        tail = p.tail;
    }
};
```

Priority Queues for Index Items

- can use array indices when elements are in an existing array

```
template <class Index>
class Priority_queue {
private:
    int N; Index* pq;
    int* qp;
    void exch(Index i, Index j) {
        int t;
        t = qp[i];
        qp[i] = qp[j];
        qp[j] = t;
        pq[qp[i]] = i;
        pq[qp[j]] = j;
    }
    void fixUp(Index a[], int k);
    void fixDown(Index a[], int k, int N);
public:
    Priority_Queue(int maxN) {
        pq = new Index[maxN+1];
        qp = new int[maxN+1];
        N = 0;
    }
    int empty() const {
        return N == 0;
    }
    void insert(Index v) {
        pq[++N] = v;
        qp[v] = N;
        fixUp(pq, N);
    }
    Index getmax() {
        exch(pq[1], pq[N]);
        fixDown(pq, 1, N-1);
        return pq[N--];
    }
    void change(Index k) {
        fixUp(pq, qp[k]);
        fixDown(pq, qp[k], N);
    }
};
```

Binomial Queues

- each of the other priority queues cannot each implement join, remove max, and insert with efficient worst case performance
    - unordered linked lists have fast join and insert, but bad remove max
    - ordered linked lists have fast remove max but slow join and insert
    - heaps have fast insert and remove max, but slow join
- efficient in this case means operations should use no more than logarithmic time
- link based heap-ordered trees require at least three links (two children, one parent)
    - also present other issues (e.g. how to join)
- one of the best solutions in this case is a binomial queue
- definition:
    - left heap-ordered: key in each node is larger than or equal to the keys in the left subtree
    - power-of-2 heap: left heap-ordered tree with an empty right subtree
    - a power-of-2 heap with a left-child, right-sibling arrangement is a binomial tree
    - binomial trees and power-of-2 heaps are equivalent (binomial trees are slightly easier to visualize)
- properties:
    - the number of nodes in a power-of-2 heaps is a power of 2
    - no node has a key larger than the key at the root
    - binomial trees are heap ordered
- for a linked representation the join is constant time
- definition:
    - a binomial queue is a set of power-of-2 heaps, no two of the same size
    - the structure of a binomial queue is determined by that queue's number of nodes
    - has a correspondence with the binary representation of integers
- NOTE: this is the answer for the previous problem showing a binary string

```
template<typename Item>
struct node {
    Item item;
    node *l;
    node *r;

    node(Item v) {
        item = v;
        l = nullptr;
        r = nullptr;
    }
};
```

```
typedef node *link;
link *bq;

static link pair(link p, link q) {
    if (p->item < q->item) {
        p->r = q->l; q->l = p; return q;
    } else {
        q->r = p->l;
        p->l = q;
        return p;
    }
}
```

- insert:

```
handle insert(Item v) {
    link t = new node(v), c = t;
    for (int i = 0; i < maxBQsize; i++) {
        if (c == 0) break;
        if (bq[i] == 0) { bq[i] = c; break; }
        c = pair(c, bq[i]); bq[i] = 0;
    }
    return t;
}
```

- get max:

```
Item getmax() {
    int i, max; Item v = 0;
    link* temp = new link[maxBQsize];
    for (i = 0, max = -1; i < maxBQsize; i++) {
        if (bq[i] != 0) {
            if ((max == -1) || (v < bq[i]->item)) {
                max = i; v = bq[max]->item;
            }
        }
    }
    link x = bq[max]->l;
    for (i = max; i < maxBQsize; i++) temp[i] = 0;
    for (i = max ; i > 0; i--) {
        temp[i-1] = x; x = x->r; temp[i-1]->r = 0;
    }
    delete bq[max]; bq[max] = 0;
    BQjoin(bq, temp);
    delete temp;
    return v;
}
```

- property:
    - all priority-queue operations can be implemented with binomial queues with O(lg(n)) operations on n-item queue

- construction of a binomial queue with n inserts requires O(n) comparisons in the worst case