Brief digression into STL — STL implementation of insertion sort : p. 263-265

```
template <typename T>
void insertionSort(vector<T>& a)
{
  int j;
  for (int p = 1; p < a.size(); ++p)
  {
      T tmp = a[p];
      for (j = p; j > 0 && tmp < a[j-1]; --j)
            a[j] = a[j-1];
      a[j] = tmp;
  }
}
```

T.operator=

⇓

type T must provide operator=

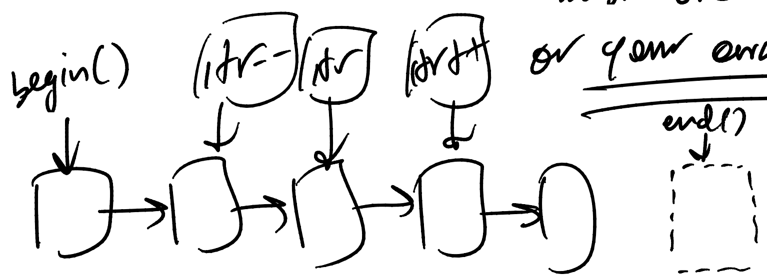
= ⇒ a.operator[](int)

< : type T must provide operator<

= : operator=

- text builds on this templated idea a bit further by providing a templated <u>iterator</u>

  like a pointer, used to iterate over a <u>container</u> (e.g. vector<>, list<>, most STL containers, or <u>your own container</u>)



Thursday, October 05, 2006
3:30 PM

```
template < typename iterator, typename T >
void insertionSort ( const iterator & begin,
{                    const iterator & end,
                     const T& obj )
```

→ text does something funny here with const T& (look it up)

```
for (iterator p = begin + 1; p != end; ++p)
{
    T tmp = *p;
    for ( iterator j = p; j != begin && tmp < *(j-1), --j)
        *j = *(j-1);
    *j = tmp;
}
```

→ (tmp < *(j-1)): book eventually generalizes this operator as well

Heapsort – best big oh running time so far

1. construct a heap of N elements $O(N)$
2. perform N deleteMin() ops $O(N \lg N)$
(each of N $O(\lg N)$)

you get this nice running time with
<u>binary heap</u>
(see text)

before this, consider
using a sorted list
for a heap
(clearly not as efficient, but a nice C++ model)

so -- consider : "listheap"

```
    template <typename T>
typename listheap<T>::iterator listheap<T>::insert(const T& dat)
{
    typename listheap<T>::iterator pp;

    for(pp = arr.begin(); pp != arr.end(); ++p)
        if(dat < (*pp)) {
            arr.insert(pp, dat);
            break;
        }
    }
    if (pp == arr.end()) arr.push_back(dat);
    return pp;
}
```

- minheap:
  - minimum (smallest) element always on top
  - findMin() operation $\in O(1)$
  - removeMin() operation: remove minimum element
    $O(\lg N)$ for binary heap
    $O(1)$, for "list heap"
  - insert() : binary heap: $O(\lg N)$
    "list heap" : $O(N)$

N elements : list heap: $O(N^2)$
binary heap : $O(N \lg N)$

Thursday, October 05, 2006
3:30 PM

- operations I'm using:
  - arr.begin()
  - arr.end()
  - arr.insert()
  - arr.push_back()

Key question: which STL container provides all of these, & which is the most efficient

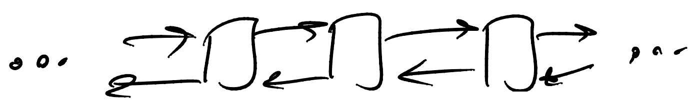
- what else will I need?
  - removeMin ⇒ arr.pop() | arr.pop_front()
  - findMin ⇒ arr.top() | arr.front()

- helpers:
  - size() : arr.size()
  - empty() : arr.empty()
  - clear() : arr.clear

Thursday, October 05, 2006
3:30 PM

vector< > :

- good performance for insert/delete at end
- possibly poor performance for insertion in the middle — may cause all elements to shift one position — O(N) operator= calls
- a <u>deque</u> may be nice — allows fast front insertion/deletion
- problem with vector<> & deque<> : storage requirements: if you exceed capacity, container will redouble its size & copy elements (hopefully) ⇒ another source of delay (potentially)
  - see vector::capacity()
    vector::reserve()
- but what we want is the STL <u>list</u>



Thursday, October 05, 2006
3:30 PM

- STL list<> has all functionality we want for our listheap, so we can use it for our "arr" data member

doing this results in something like an "adapter" design pattern → our listheap looks like STL list, but has added functionality / properties: the list is always sorted (STL list doesn't guarantee this)

```
#include <list>
using namespace std;   // local!
// forward declarations
template <typename T> class listheap;
template <typename T> ostream & operator<<(ostream& s,
                                           listheap<T>& rhs);

template <typename T>
class listheap {
        private:
         list<T>           arr;
        public:
         listheap()        { };  // constructor — call
                                 //   appropriate one
        typedef typename list<T>::iterator   iterator;
                                               ↓
// typename list<T>::iterator  <--means--  typename listheap<T>::iterator
    int size() const             { return arr.size(); }
   bool empty() const            { return arr.empty(); }
   void clear()                  { arr.clear(); }
  const T& top() const           { return arr.front(); }
   void pop()                    { arr.pop_front(); }
  iterator begin()               { return arr.begin(); }
  iterator end()                 { return arr.end(); }
  iterator insert(const T&);
  const T& at(iterator i) const  { return *i; }
  T& at(iterator i)              { return *i; }
                                     ↑ if (i != arr.end())
```

Usage: #include <sys/types.h> or unsigned long

```
listheap<float>   heap;
listheap<float>::iterator   heapi;
srandom((ulong) 0x1337)
for(k=0; k<10; ++k) {
    no = random() / RAND_MAX * 10.0
    heap.insert(no)
}
heap.insert(6.0)
heap.insert(9.0)

heapi = heap.find(6.0)
heap.at(heapi) = 12.0;  // oops! just munged
                        list order
heap.inckey(heapi);
```

tell the heap that key (value) at pos. heapi was incremented

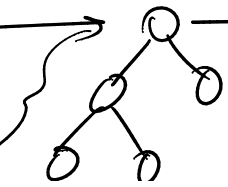
⇒ result: node at heapi pos. gets removed & shifted down

```
while(!heap.empty()) {
    std::cout << "deleting min: " << heap.top();
    heap.pop();
}
}
```

analogous to "percolate down" of binheap

- binary heap notes:
  - binary tree used as underlying ADT

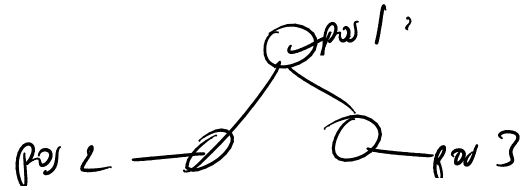
top() O(1) → min element

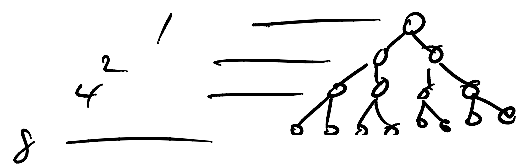


children's "keys" are > their parents'

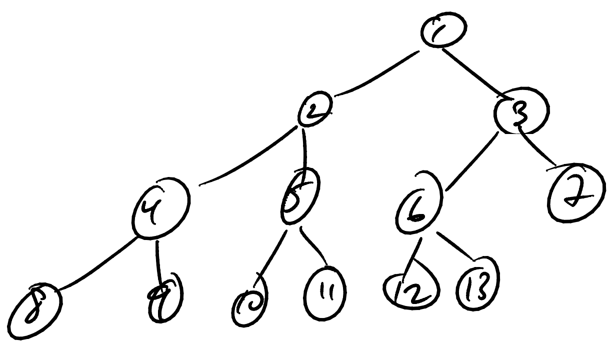
pop(): O(lg N) — will remove root, & fill in its "hole" with some node down the tree (lg N height)

- binary heap is actually implemented as array — uses a vector<> (pos 0 not used)

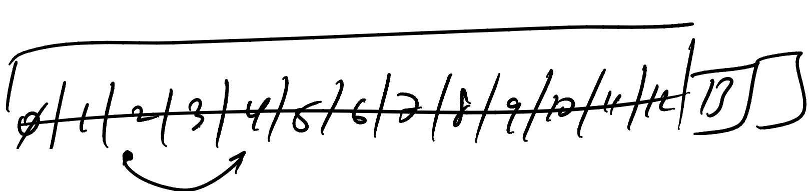




child = parent's pos ×2



there are just indices, not keys



- the min heap property:
  min element always at root
  is preserved via insert() & pop() operations.
  percolateUp(), percolateDown()
- AS65: basically same driver as "listheap",
  except: what is binheap<T>::iterator?
  just an int — position of element on list.