CS2100

Sets

Recap

- Work day tomorrow (on HW & lab)
- Practice Problems sheet is also available (& practice final)
- Double check grades in BB (& on git)
  - Still 2 HWs to grade...
- Review Monday & final next Wed. at 2pm

Sets: The set ADT supports several functions:

- insert(e): insert element e into the data structure
- find(e): return iterator (or T/F) if e is in set
- erase(e): remove e from S
- begin/end: return iterator to beginning/ending

Also (in STL):

- lower_bound(e): return iterator to the largest element $\leq e$
- upper_bound(e): // ($\geq e$)
- equal_range(e): return range of elements == e iterator

Issue: can e be in set ≥ 1 time?

STL

```
// set::lower_bound/upper_bound
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;
  std::set<int>::iterator itlow,itup;

  for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90

  itlow=myset.lower_bound (30);                //        ^
  itup=myset.upper_bound (60);                 //                       ^

  myset.erase(itlow,itup);                     // 10 20 70 80 90

  std::cout << "myset contains:";
  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
    std::cout << ' ' << *it;
  std::cout << '\n';

  return 0;
}
```

Notice that `lower_bound(30)` returns an iterator to `30`, whereas `upper_bound(60)` returns an ite

```
myset contains: 10 20 70 80 90
```

Example

```
// set::count
#include <iostream>
#include <set>

int main ()
{
  std::set<int> myset;

  // set some initial values:
  for (int i=1; i<5; ++i) myset.insert(i*3);    // set: 3 6 9 12

  for (int i=0; i<10; ++i)
  {
    std::cout << i;
    if (myset.count(i)!=0)
      std::cout << " is an element of myset.\n";
    else
      std::cout << " is not an element of myset.\n";
  }

  return 0;
}
```

Output:

```
0 is not an element of myset.
1 is not an element of myset.
2 is not an element of myset.
3 is an element of myset.
4 is not an element of myset.
5 is not an element of myset.
6 is an element of myset.
7 is not an element of myset.
8 is not an element of myset.
9 is an element of myset.
```

How to implement?

- List - probably sorted
  - ↳ private:

```
List<T> myData;
// functions can use
// list helper functions
```

- Vector
  (same)

- AVL tree

STL: balanced BST
↳ red-black tree

Trade-offs:

Space: $O(n)$

Functions:

(balanced search) Tree:

all $O(\log n)$

Vector:

insert: $O(n)$

↳ search $O(\log n)$

↳ insert in vec $O(n)$

find: $O(\log n)$

List:

insert: $O(n)$ (if sorted)

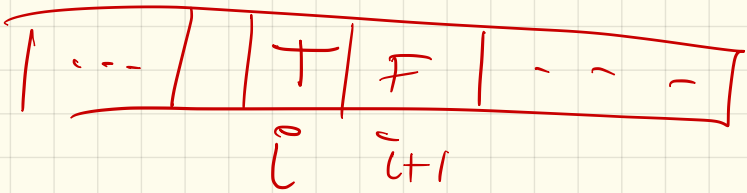
find: $O(n)$

One other option:

Bit Vector

bool values[max_value];

If values in set range from 0 to N-1



Downside: space!

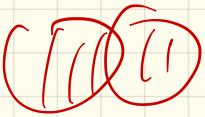
With n values, size is still O(N).

Lookup, insert + remove: O(1)

Mergable Sets:

Adds:

- union (S1, S2): returns a new set containing all elements in S1 or S2



Note: could also merge the two objects

- interset (S1, S2):

- setminus (S1, S2):

Runtimes

depends!

Vector or List: O(n)

Special case: Union-find

Only 3 operations

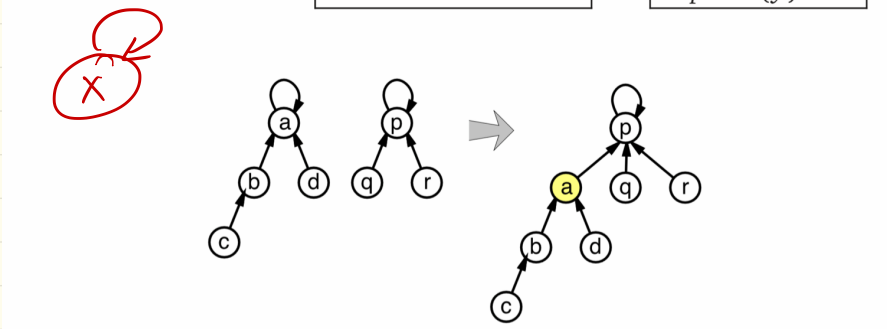
- makeSet(x): create a set containing only x
- union(A, B): return A ∪ B (& destroy A & B)
- find(p): return the set containing p.

Implementation:

```
MAKESET(x):
    parent(x) ← x
```

```
FIND(x):
    while x ≠ parent(x)
        x ← parent(x)
    return x
```

```
UNION(x, y):
    x̄ ← FIND(x)
    ȳ ← FIND(y)
    parent(ȳ) ← x̄
```



- Each set needs to "know" its component
- Initially, each set is its own -> n labels
- When combining 2,
  - take smaller set & relabel all of its vertices

(See prev. slide)

- Then, each time a component label changes, its set is ≥ twice as large.

So: each label can change only $O(\log n)$ times

(used for MST algorithm)