Using the Standard Template Library (STL)

The Standard Template library

- developed by Alexander Stepanov
- part of the official C++ standard in 97
- most notable example of generic programming and abstractness
- very efficient
- This lecture is about using and applying the STL in your own codes.

The Standard Template library

- containers: wrappers for data, can be sequential (vector, list, deque), associative (map, set, hash) or adaptive (stack, queue — C++11)

- iterators: are the major feature that allow the generality of the STL. There are 5 types: *input*, *output*, *forward*, *bidirectional* and *random access* iterators

- algorithms: typically searching and sorting (binary search, lower_bound, …) and often requires a custom operator < which must guarantee a strict weak ordering

- functions: certain classes can overload the function call operator(). Such instances are known as *functors*. A typical example is a *predicate* (= a boolean valued function) used eg in find_if : it takes a unary predicate that operates on the elements of a sequence

- allocators: used for dynamic memory allocation when the size of the containers changes. This will all occur internally and be of no concern in this lecture, but you can use the STL allocators to allocate memory for your own objects

O(N) notation

- when analyzing data structures and algorithms we are interested in
  - how much CPU time is required?
  - how much memory is required?
  - how big is the bandwidth (transfer of data)?
- the crucial question is how these numbers scale when increasing (eg doubling) the input size: constant, polynomially or exponentially? A difference between worst-case, best-case and typical scenario can also be made

O(N) notation

- example : addition as taught in primary school

```
   101
   215
   846
+ ----
  1061
```

size of input : n digits per input number

however there are also n carry digits

so we have to perform 2n additions

time required: $T(n) = 2n$

summing is a linear function in the length of the input

O(N) notation

- example : multiplication as taught in primary school

$$
\begin{array}{r}
151 \\
\times \quad 175 \\
\hline
755 \\
1057 \\
+ \; 151 \\
\hline
26425
\end{array}
$$

size of input : n digits per input number

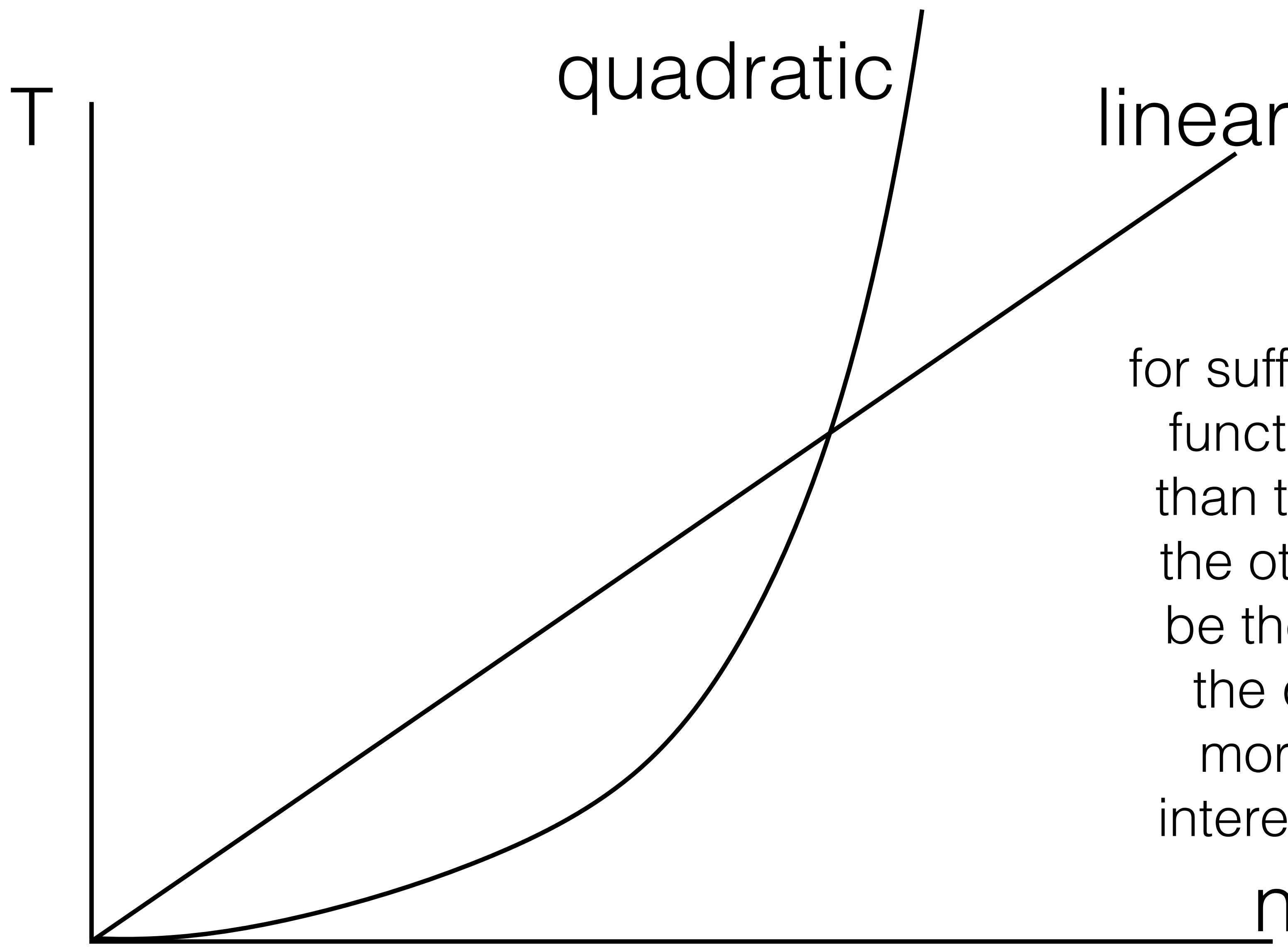
number of multiplications : $n \times n = n^2$

number of additions : also proportional to n^2

time required: $T(n) = n^2 + an^2 = cn^2$

multiplication is a quadratic function in the input

O(N) notation

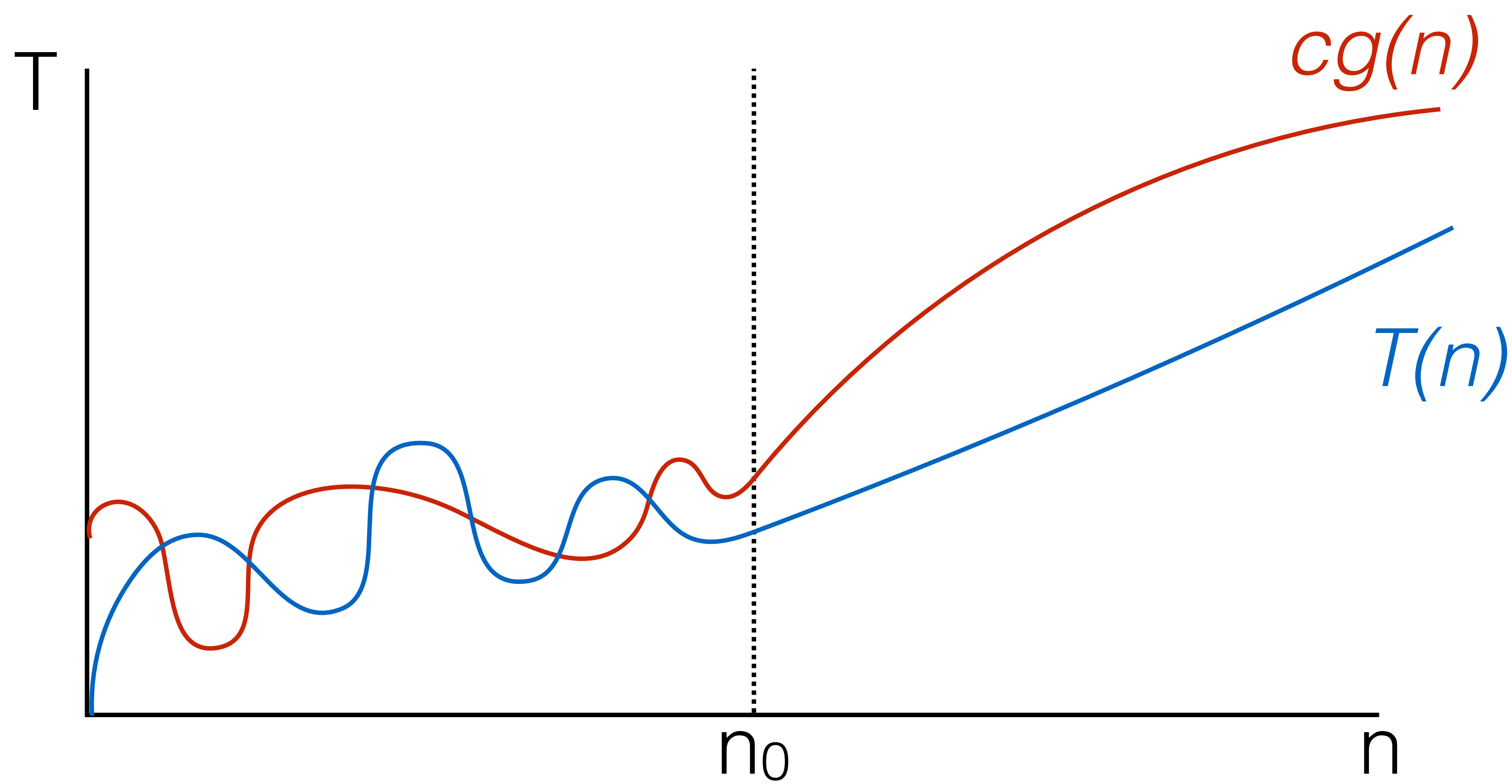


for sufficiently large n the quadratic function takes always more time than the linear one, for small n on the other hand the overhead can be the dominant factor such that the quadratic function may be more efficient. We are usually interested in the large n behavior

O(N) notation

definition:

the function $T(n) = \mathcal{O}(g(n))$ when there exist constants *c* and n_0 such that $T(n) \leq cg(n) \;\; \forall n > n_0$



note: mathematicians also use the little-o notation: *T(n) = o(g(n))* which holds when there exists a constant n_0 such that $T(n) \leq \epsilon g(n) \;\; \forall n > n_0$ for *arbitrary* ε. It implies that the ratio T(n) / g(n) vanishes in the large n limit

O(N) notation

examples:

$$T(n) = n^2 = \mathcal{O}(n^2)$$

$$T(n) = n^2 + 2n = \mathcal{O}(n^2)$$

$$T(n) = n^2 + \log(n) = \mathcal{O}(n^2)$$

$$T(n) = n^2 + 2n = \mathcal{O}(n^3)$$

since big-O only introduces an upper bound, computer scientists introduce the following 2 definitions:

complexity analysis

definition (for best case analysis):

the function $T(n) = \Omega(g(n))$ when there exist constants *c* and n_0 such that $T(n) \geq cg(n) \;\; \forall n > n_0$

definition (if best and worst case are the same):

the function $T(n) = \theta(g(n))$ when $T(n) = \mathcal{O}(g(n))$ and $T(n) = \Omega(g(n))$

quite often, the big-O notation is used where really the θ-notation is meant!

assuming 10 GFlop (~ 2.5 GHz processor) and an operation takes about 0.1 ns

complexity	N=10	10^2	10^3	10^4	10^5	10^6
1	0.1 ns	0.1 ns	0.1 ns	0.1 ns	0.1 ns	0.1 ns
log N	0.3 ns	0.7 ns	1.0 ns	1.3 ns	1.7 ns	2.0 ns
N	1 ns	10 ns	100 ns	1 μs	10 μs	0.1 ms
N logN	3.3 ns	66.4 ns	1 μs	13 μs	0.17 ms	2 ms
N^2	10 ns	1 μs	0.1 ms	10 ms	1 s	1.7 min
N^3	0.1 μs	0.1 ms	0.1 s	1.7 min	> 1 day	> 3 years
2^N	0.1 μs	10^{14} years	10^{285} years			

A little quiz

consider two matrices of size NxN. What is the complexity of their multiplication?

A. quadratic $\theta(N^2)$
B. cubic $\theta(N^3)$
C. sub-cubic $\theta(N^p), p < 3$
D. exponential $\theta(c^N)$

and what is the complexity of a matrix-vector multiplication?

is there a difference between the computer-time and storage complexity in any of these cases?

A little quiz

we compute the fibonacci numbers recursively

```
double fib(unsigned int n) {
  if (n == 1) return 1;
  if (n == 0) return 0;
  return (fib(n-1) + fib(n-2));
}
```

what is the complexity of this algorithm?

A. constant $\theta(1)$
B. linear $\theta(N)$
C. quadratic $\theta(N^2)$
D. exponential $\mathcal{O}(c^N)$

is there a faster way?

A little quiz

consider the (unnormalized) discrete Fourier transform of data *f(j)* of length N

$$F(k) = \sum_{j=0}^{N-1} e^{2\pi\sqrt{-1}jk/N} f(j)$$

what is the complexity of computing Fourier transforms?

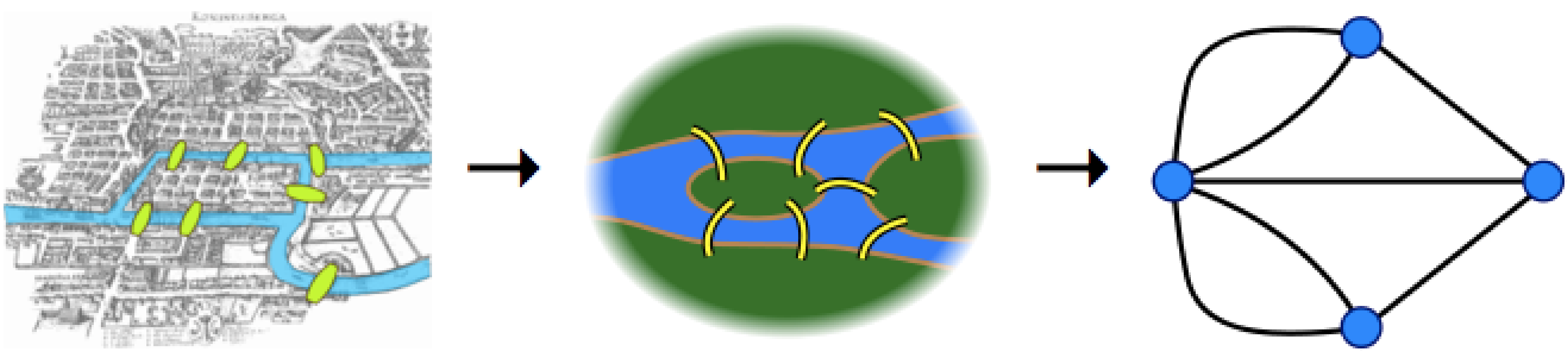
A. constant $\theta(1)$
B. linear $\theta(N)$
C. quadratic $\theta(N^2)$
D. other

Something to think about

- what is the complexity of finding a number in an array of numbers?
- what is the complexity of finding a name in a telephone book?
- what is the complexity of sorting all student names alphabetically?

Something to think about

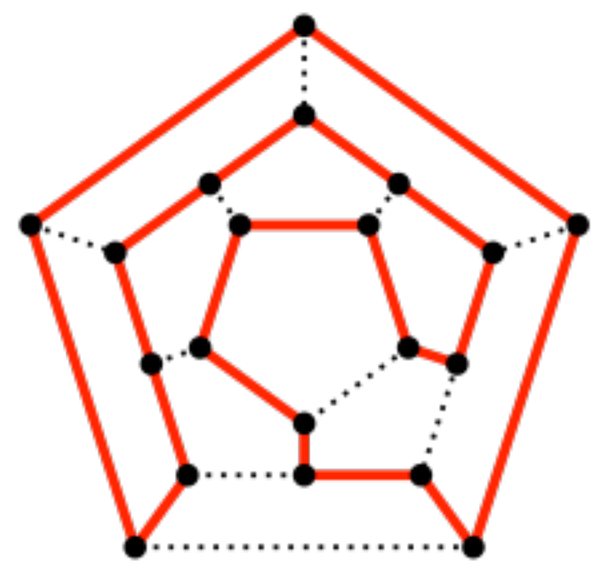
Eulerian circuit problem



- 7 bridges of Königsberg
- is there a roundtrip that crosses each bridge (*edge*) exactly one time?
- more general: what is the complexity class of the best algorithm you can think of for N bridges?

Something to think about

Hamilton cycle problem



- is there a path that crosses each *vertex* exactly once?
- more general: what is the complexity of the best algorithm you can think of for N *vertices*?

Standard containers

- pair : a tuple of 2 elements (general tuples exist since C++11)
- array structures : vector, deque, valarray
- list : list
- tree : map, multimap, set, multiset
- queue : queue, priority_queue, stack
- string and wstring

Since C++11 there is support for a hash table implemented in the unordered_map and the unordered_set (see also the lecture on C++11)

pair

- a container for a pair of elements of possibly distinct type

```
template <class T1, class T2> class pair {
public:
    T1 first;
    T2 second;
    pair(const T1& f, const T2& s) : first(f), second(s) {}
};
```

an important function is make_pair:

```
template <class T1,class T2>
pair<T1,T2> make_pair (T1 x, T2 y)
{
  return ( pair<T1,T2>(x,y) );
}
```

example modified from cplusplus.com

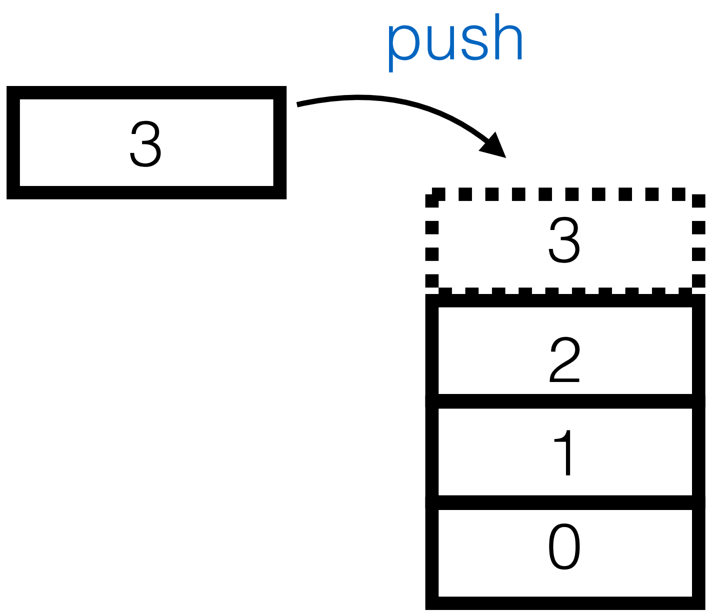
```
// make_pair example
#include <utility>      // std::pair
#include <iostream>     // std::cout

int main () {
  std::pair <int,int> foo;
  std::pair <int,int> bar;
  std::pair <int,char> baz;

  foo = std::make_pair (10,20);
  bar = std::make_pair (10.5,'A'); // ok: implicit conversion from pair<double,char>
  baz = std::make_pair (10.5,'A');

  std::cout << "foo: " << foo.first << ", " << foo.second << '\n';
  std::cout << "bar: " << bar.first << ", " << bar.second << '\n';
  std::cout << "baz: " << baz.first << ", " << baz.second << '\n';

  return 0;
}
```

String and wstring classes

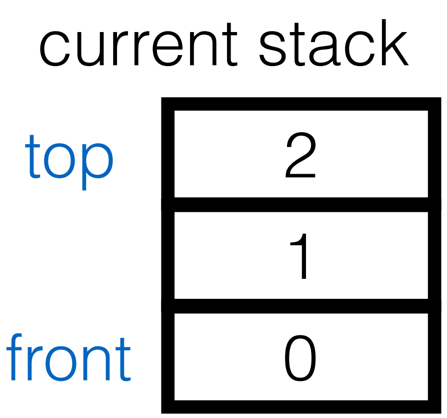
is usually not so important for numerical work
has partly been addressed in the first lecture;
will however be addressed again when
discussing streams
read your C++ book

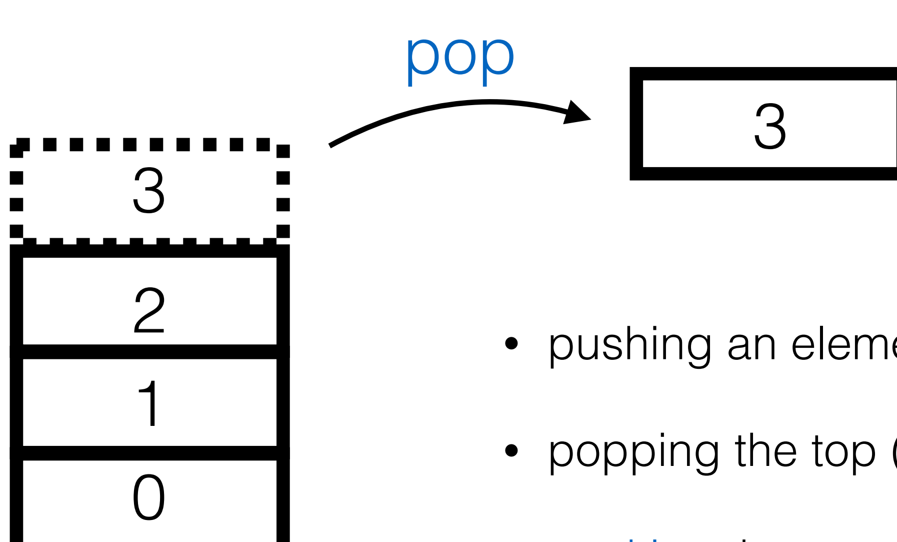
stack data structure: LIFO

- last-in first-out or last-come first-serve



example : piled-up trays at the mensa





- linear data structure
- single pointer to top of the stack needed
- peek : retrieve top element without removing it

- pushing an element to the top is an O(1) operation
- popping the top (=last) element is an O(1) operation
- peeking the top element is an O(1) operation

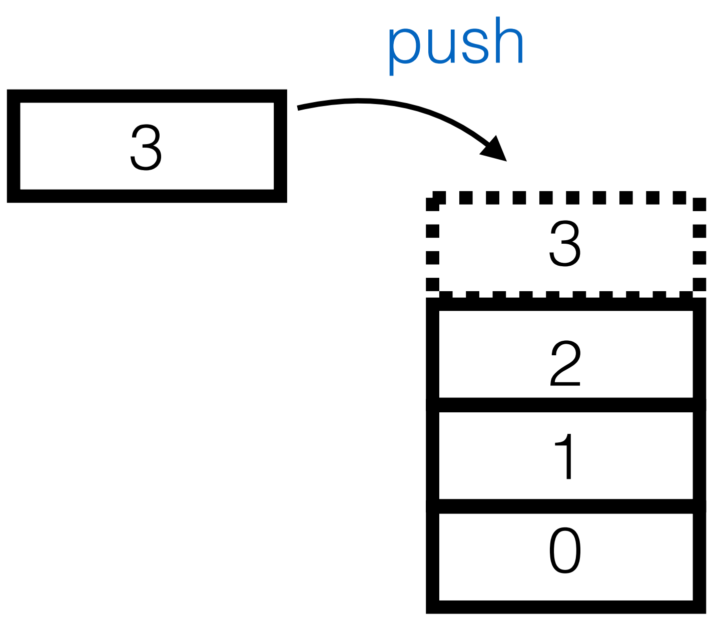
stack

- implementation of a LIFO structure
- functions : empty, size, top, push, and pop

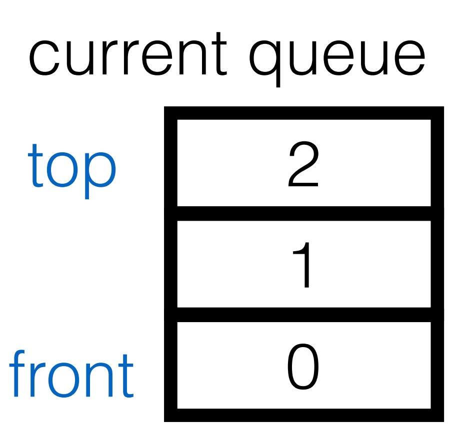
```
// example from cplusplus.com : stack::push/pop
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main ()
{
  std::stack<int> mystack;

  for (int i=0; i<5; ++i) mystack.push(i);

  std::cout << "Popping out elements...";
  while (!mystack.empty())
  {
     std::cout << ' ' << mystack.top();
     mystack.pop();
  }
  std::cout << '\n';

  return 0;
}
```

queue data structure: FIFO

- first-in first-out or first-come first-serve

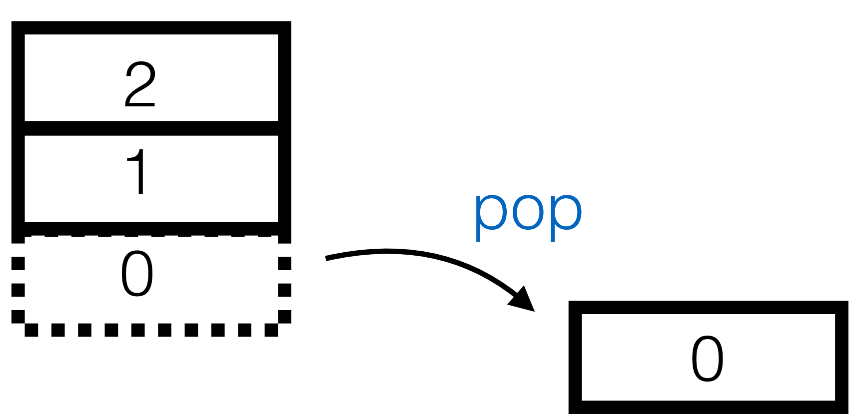


example : waiting line



- pushing an element to the top is an O(1) operation
- popping the first element is an O(1) operation
- peeking the first and last element is an O(1) operation

- linear data structure
- single pointer to top of the stack needed
- peek : retrieve top element without removing it



queue

- implementation of a FIFO structure

- functions : empty, size, front, back, push, and pop

```
#include <iostream>       // std::cout
#include <queue>          // std::queue

int main ()
{
  std::queue<int> myqueue;

  myqueue.push(100);
  myqueue.push(20);
  myqueue.push(10);

  myqueue.front() -= myqueue.back();    // 100 - 10 = 90

  std::cout << "myqueue.front() is now " << myqueue.front() << '\n';
  for (int i=myqueue.size()-1; i >=0; i--) {
      std::cout << myqueue.front() << "\n";
      myqueue.pop();
  }
  return 0;
}
```

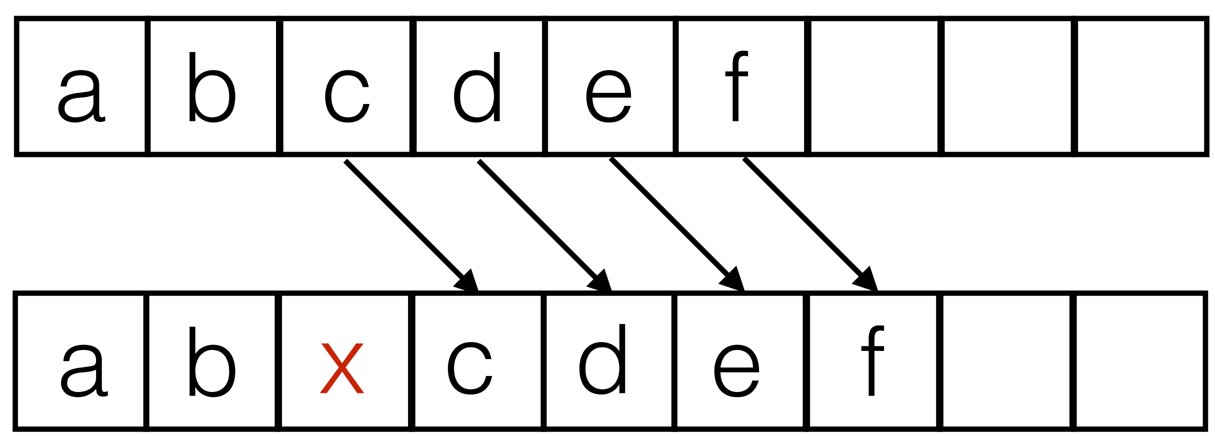
- there also exists a priority_queue : a queue in which the first element is always the greatest element following a strict order (see documentation)

vector

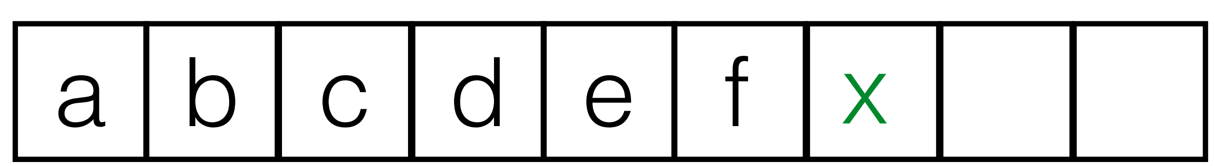
http://www.cplusplus.com/reference/vector/vector/

- is a sequence container that can change in size
- use contiguous storage locations for their elements
- size can change dynamically; vectors may allocate extra storage, *so the capacity may be larger than its size*
- fast lookup of its elements
- relatively efficient adding/removing elements from the end
- slow in inserting/removing elements at other positions

slow O(N) insertion and removal: when inserting an element in the middle, all the elements to the right must be copied and moved



amortized O(1) for insertion/ removal at the end (spare capacity!)



see ex_vector.cpp for its usage and more concepts

the std::vector implements what computer scientists understand under the array (aka vector) data structure

note that the specialization vector<bool> may behave unexpectedly, see http://www.cplusplus.com/reference/vector/vector-bool/

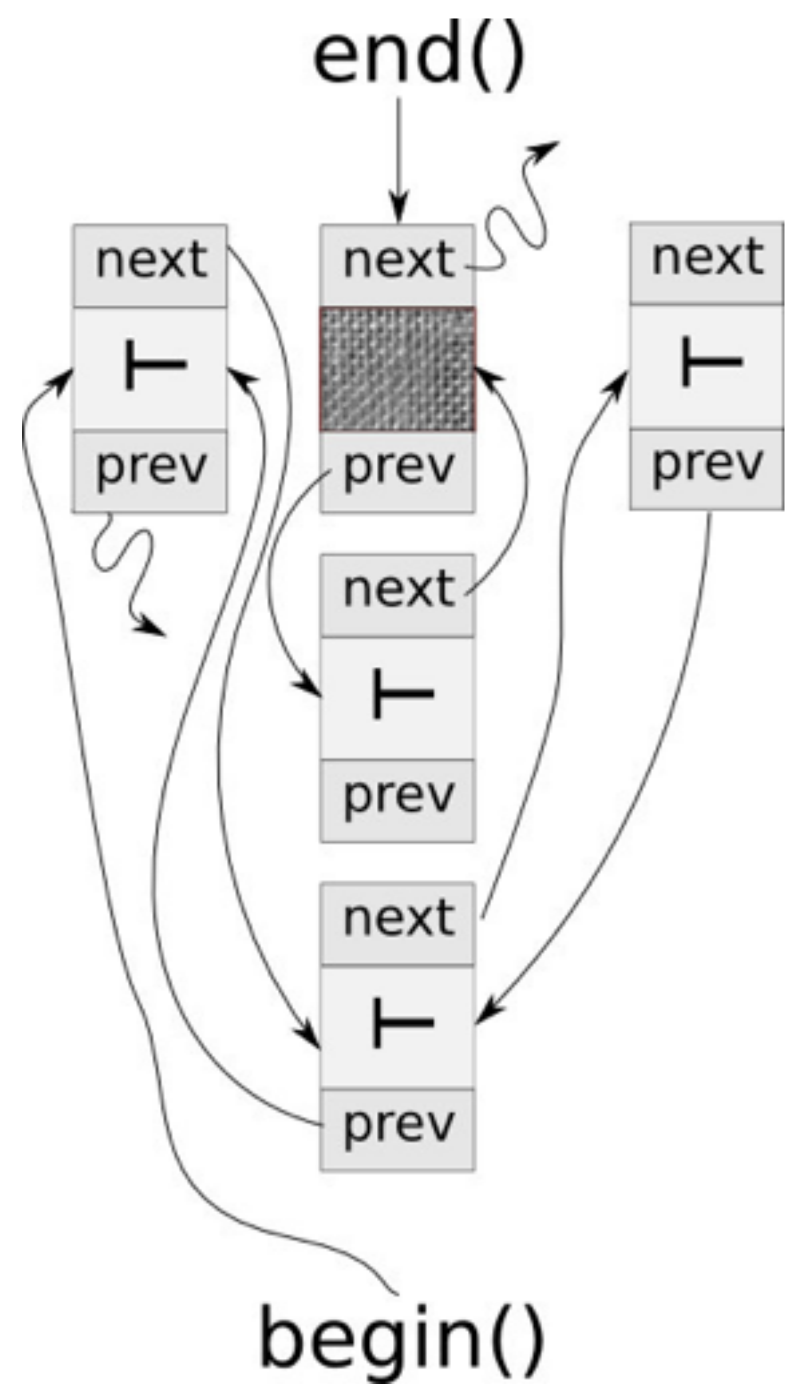
deque

- pronounced “deck”
- is a **D**ouble-**E**nded **Que**ue
- efficient insertion and deletion at the beginning and at the end of the data sequence
- does NOT store data in contiguous memory (hence pointer arithmetic is undefined)
- has random access iterators

list

http://www.cplusplus.com/reference/list/list/

- is a sequence container that can change in size; not contiguous in memory
- O(1) insertion and removal everywhere in the list
- doubly-linked (the forward list is singly-linked)
- lacks direct access to the elements by its position



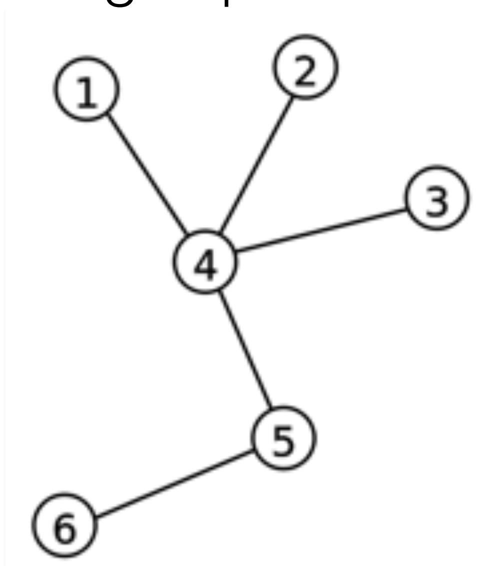
see ex_list.cpp for its usage and more concepts

Tree structures

- an array needs
    - O(1) for random access
    - O(N) for arbitrary insertions and removals
    - O(N) for searches
    - O(log N) for searches in a sorted array
- A list needs
    - O(1) for arbitrary insertion and removal
    - O(N) for random access and searches
- what if both operations need to be fast? Use tree data structure
    - O(log N) for arbitrary insertion and removal
    - O(log N) for random access and searches

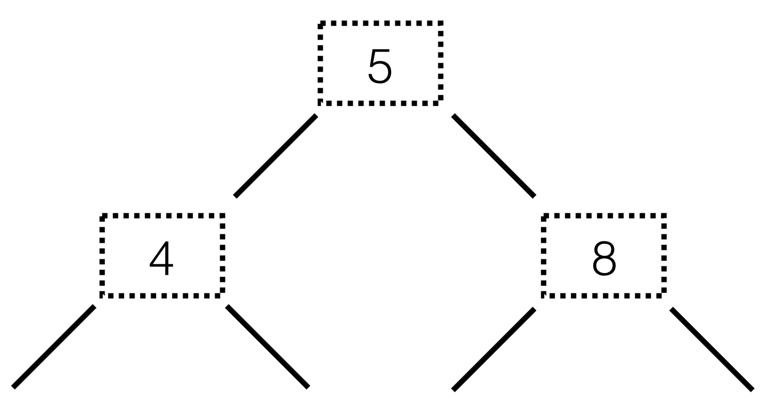
Tree structures

- A tree is an undirected graph in which any 2 vertices are connected by exactly one path
- in computer science rooted trees are important: one designated vertex from which all edges point away



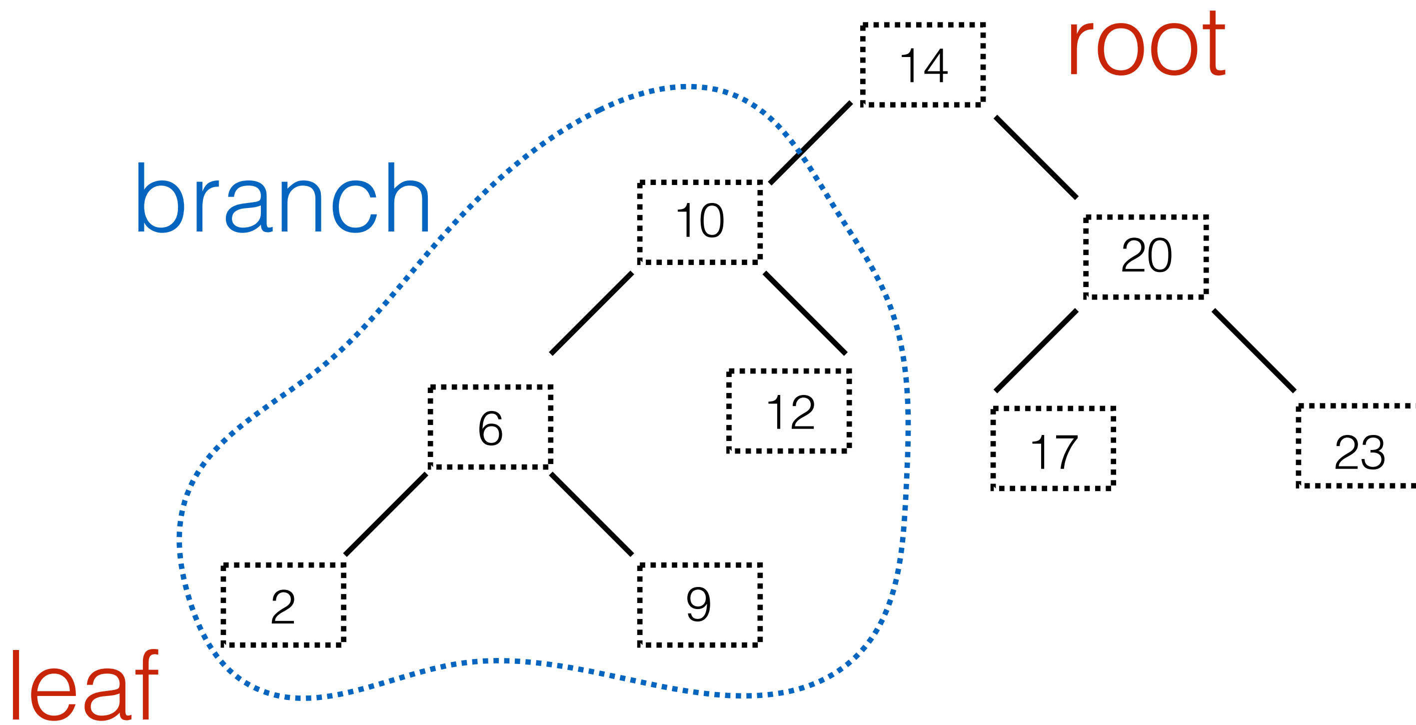
example of a tree
there are n vertices, n-1 edges

- A binary tree : every node has (at most) 2 child nodes, the right child node is larger than the parent and the left child node

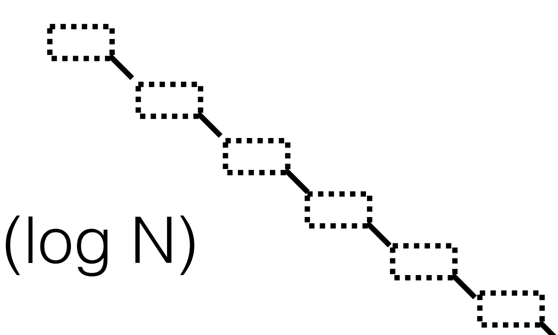


binary tree

- a tree of height n can store $N = 2^n - 1$ elements
- if the tree is perfectly balanced, then a search can take no more than n operations



- trees can however become unbalanced:
  imagine inserting elements from a sorted array:

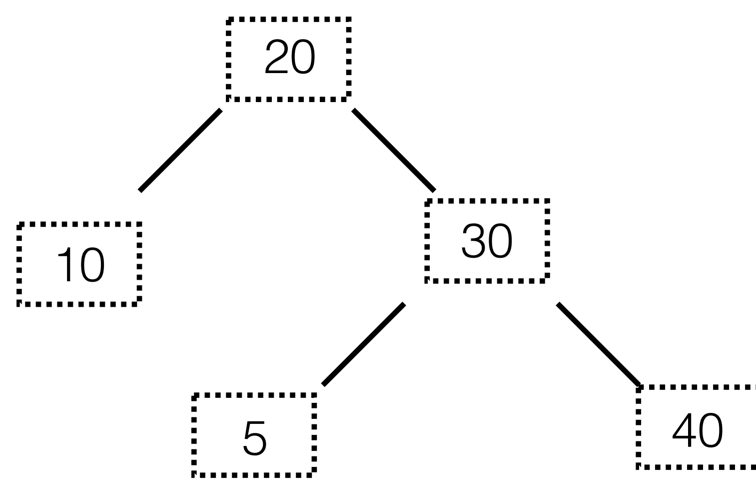


this is problematic because all operations become O(N) instead of O(log N)

solution : rebalancing methods (eg rotations), or use self-balancing trees

binary search tree

- in a binary search tree all elements on a right subtree must be larger than those on any parent node along the subtree



is not a BST

- search : start at root, compare value, go left or right etc
- insert: start with search, then insert new key-value pair as new leaf
- delete: a bit more complicated (the node to be deleted may have dependencies) — see textbooks

see textbooks on computer science for more tree structures (red-black tree, 2-3-4 tree, …

map

- are associative containers storing elements as pairs <key, value>. The keys are usually used to sort (following a strict weak ordering criterion) and identify the elements *uniquely*

- are implemented as binary search trees

- has the usual iterators: begin, end, rbegin, rend, …

- has functions : insert, erase, clear, empty, size, …

- has operator[] to access the elements — map[key] returns map_value

- has operations: find, count, lower_bound, upper_bound, equal_range

map example

- see the example ex_map.cpp in this week's programs for usage

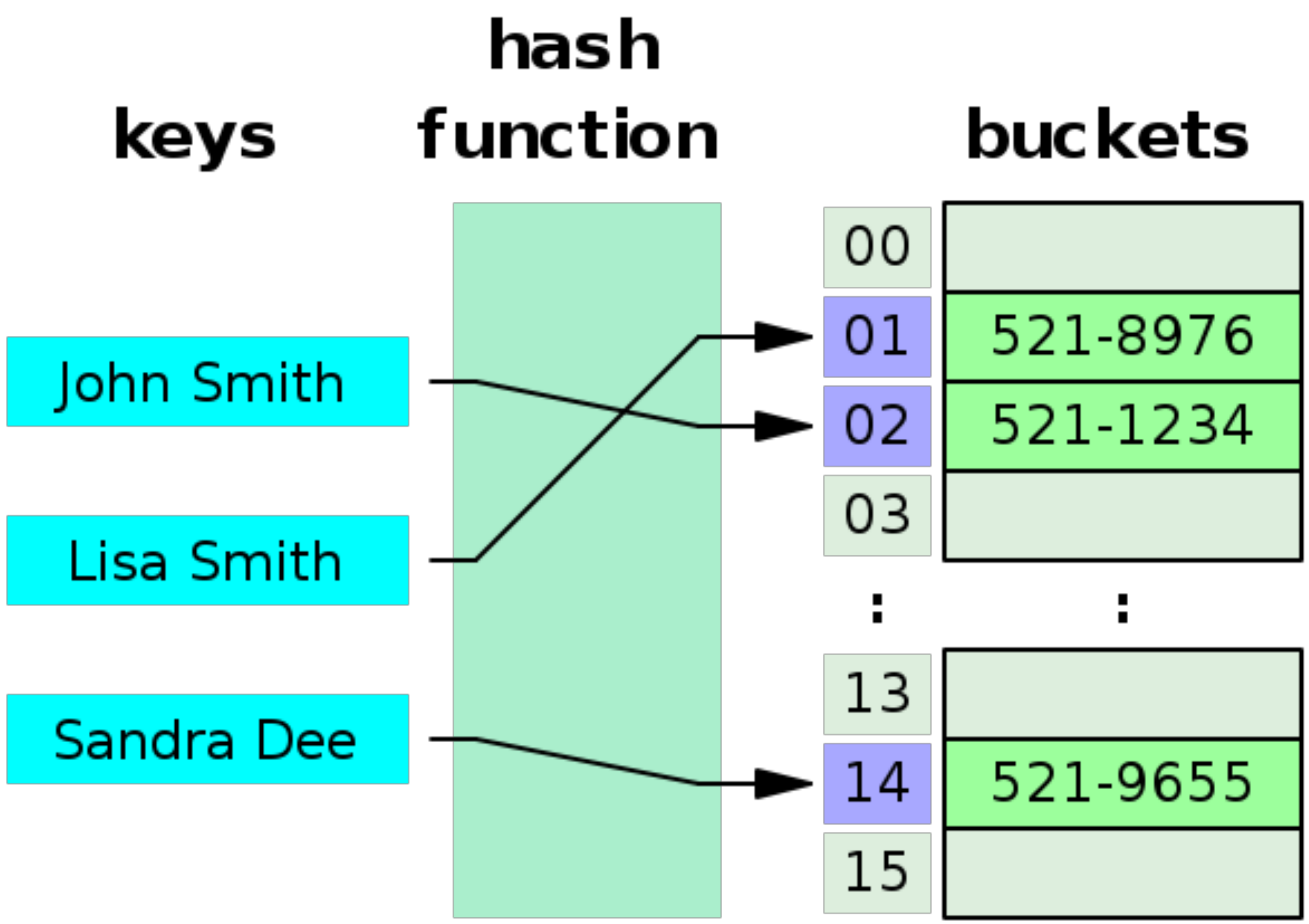
- multimap : key does not have to be unique (see the documentation)

- set : ordering is based on values only; value must be unique (see the documentation)

- multiset : like set, but value must not be unique

- do not be confused: the unordered_map, unordered_multimap, unordered_set, unordered_multiset (C++11) : is not ordered internally, but uses a *hash* function

hash table



Hash table

Type	Unordered associative array	
Invented		1953
	Time complexity	
	Average	Worst case
Space	O(*n*)[1]	O(*n*)
Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)

```
index = f(key, array_size)
```

often done in two steps:

```
hash = hashfunc(key)
index = hash % array_size
```

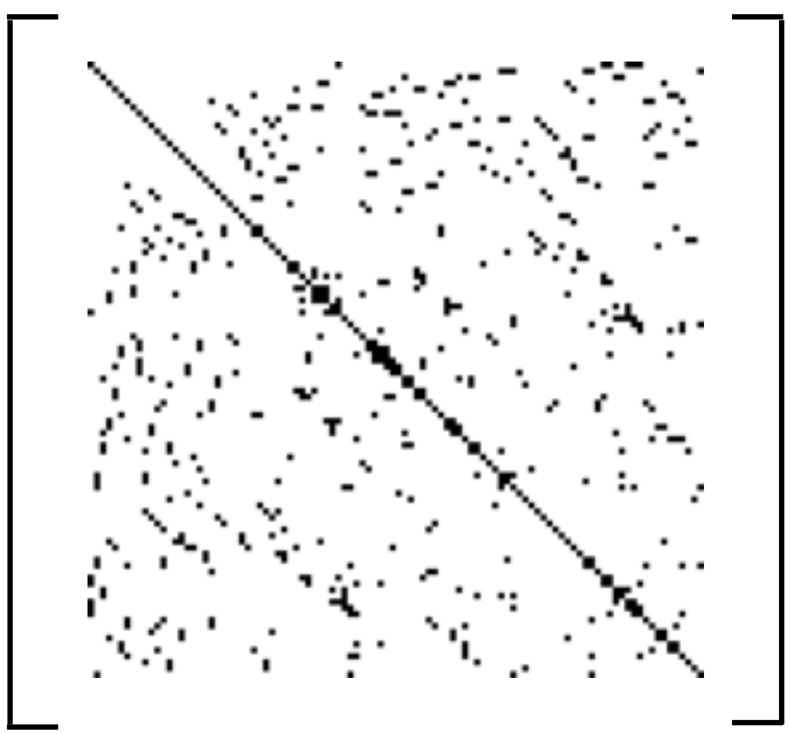
a good hash gives a good idea where the index can be found

hash table

- typical situation in physics: diagonalization. The Hamiltonian is stored as a sparse matrix

most elements are zero!

H =



how to store? many options, for instance a coordinate list COO with tuples (row, column, value) sorted first by the row index, then by the column index

when H acts on a vector, we need to know the *index* of the final state

how does a hashing function work?

- consider the exact diagonalization of a spin-1/2 system. Let there be L sites and consider the magnetization sector M = 0 — ie, we assume that the S^z operator commutes with the Hamiltonian.

- without conservation of the magnetization there would be 2^L states
- in the submanifold there are however fewer states: $\frac{L!}{(L/2)!(L/2)!}$

- states can be represented by bits:

$$\uparrow \rightarrow 1 \qquad \downarrow \rightarrow 0$$

- a state can hence be represented as an integer (this is a bijection):

$$[1, 0, 0, 1] = 9$$

- this allows for fast bit operations via AND, OR and XOR

how does a hashing function work?

- for L = 4, there are 16 states in total and 6 states with magnetization 0

$$[0,0,1,1] = 3$$
$$[0,1,0,1] = 5$$
$$[0,1,1,0] = 6$$
$$[1,0,0,1] = 9$$
$$[1,0,1,0] = 10$$
$$[1,1,0,0] = 12$$

we can iterate over all 2^L states and check the magnetization; alternatively, we start with the state with lowest possible integer representation and attempt to move iteratively the right-most 1-bits to the left until we reach another 1 or the end of the sequence (see exercises)

- we only want to store these 6 valid states. The problem now is to find the index of a valid state. Eg, given [1001] = 9 we need to know that it has index 3.
- solution nr 1: use a binary search tree (std::set or std::map) or a sorted vector. In this case the lookup is logarithmic in the the number of states (O(log N)) which in most cases is fast enough.

how does a hashing function work?

- solution nr 2: with a hashtable we can provide lookup with O(1). This can be done with std::unordered_multimap. To show how hashing works, we will implement our own hash function

state	index	hash
$[0,0,1,1] = 3$	0	3
$[0,1,0,1] = 5$	1	5
$[0,1,1,0] = 6$	2	0
$[1,0,0,1] = 9$	3	3
$[1,0,1,0] = 10$	4	4
$[1,1,0,0] = 12$	5	0

- a simple, though not the optimal, hash function is hash = (state % 6)

how does a hashing function work?

- we can now store two arrays. The first one consists of <key, value> pairs where key is the result of the hash function and value the decimal representation of the state:

$$[<0,2>,<0,5>,<3,0>,<3,3>,<4,4>,<5,1>]$$

- one sees that the keys are not unique because our hash function is not perfect. In the above example there are 2 collisions
- the above array does not need to be ordered, but entries with the same keys (‘buckets’) must be consecutive.
- the second array is the index array H:

$$[0, nan, nan, 2, 4, 5]$$

- H[n] = p it tells us the location of the first entry in the <key, value> array where the key is p.
- nan means here that 1 and 2 cannot be valid results of our hash function over the chosen Hilbert space

example

- we wish to know the index of [1, 0, 1, 0]

  decimal : 10 ; hash result : 4.
  result of index array : 4. The corresponding value to this key is unique, namely 4; ie the index of [1,0,1,0] is 4.

- we wish to know the index of [1, 0, 0, 1]

  decimal : 9 ; hash result : 3.
  result of index array : 2. The corresponding value to this key is not unique, both 0 and 3 are valid. Then we need to check explicitly which state we want.
  After comparing the 2 values with the initial state, we deduce that its index is 3

- for large Hilbert spaces and very good hash functions this can produce a lookup with O(1) complexity.
- it functions best when the manifold of states remains invariant (ie inserts and removes are seldom)
- a good hash function is usually given by (. % p) with p the smallest prime number larger than the size of the Hilbert space.
- in the exercises you will work out this example

unordered_map

```
// unordered_map --  requires C++11
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
  std::unordered_map<std::string,int> bachelor_students, all_students;

  std::pair<std::string,int> exchange_student ("Fabricio", 2085);
  all_students.insert(exchange_student);
  bachelor_students.insert(std::make_pair<std::string,int>("Ann", 2183));
  bachelor_students.insert(std::make_pair<std::string,int>("Tom", 2184));
  all_students.insert(bachelor_students.begin(), bachelor_students.end());

  std::cout << "all_students contains " << std::endl;
  for (std::unordered_map<std::string,int>::iterator it = all_students.begin(); it != all_students.end(); ++it) {
    std::cout << it->first << " : " << it->second << std::endl;
  }

  std::cout << std::endl;

  std::unordered_map<std::string,int>::hasher hfun = all_students.hash_function();
  for (std::unordered_map<std::string,int>::iterator it = all_students.begin(); it != all_students.end(); ++it) {
    std::cout << hfun(it->first) << std::endl;
  }

  return 0;
}
```

STL algorithms

- The STL defines a huge number of algorithms that work on almost all containers in <algorithm> :

  see http://en.cppreference.com/w/cpp/algorithm for a complete list

- a few examples:
  - count : counts how often a predicate is true over a sequence

    ```
    int num_items1 = std::count(v.begin(), v.end(), target1);
    ```

  - find : searches for an element equal to value

    ```
    std::vector<int>::iterator it = std::find(v.begin(), v.end(), 3);
    ```

  - copy : copies a range

    ```
    std::vector<int> to_vector;
    std::copy(from_vector.begin(), from_vector.end(),
              std::back_inserter(to_vector));

    std::cout << "to_vector contains: ";

    std::copy(to_vector.begin(), to_vector.end(),
              std::ostream_iterator<int>(std::cout, " "));
    std::cout << '\n';
    ```

STL algorithms

- swap : swaps the contents of 2 elements

```
int a = 5, b = 3;
std::swap(a,b);
```

- unique : removes duplicate elements from a sequence

```
int myints[] = {0,7,5,6,1,2,2,3,4,7,4,3,3,3};
std::vector<int> v (myints, myints + sizeof(myints) / sizeof(int) );
std::sort(v.begin(), v.end()); // 0 1 2 2 3 3 3 3 4 4 5 6 7 7
std::vector<int>::iterator last = std::unique(v.begin(), v.end()); // 0 1 2 3 4 5 6 7 x x x x x x
v.erase(last, v.end());  // remove indeterminate elements
```

- sort : sorts a range in ascending order
- lower_bound : (on sorted range): returns an iterator to the first element not less than the given value
- upper_bound : (on sorted range): returns an iterator to the first element greater than the given value

STL algorithms

- max : returns the greater of two given values
- max_element : returns the largest element in a range
- min : returns the smaller of two given values
- min_element : returns the smallest element in a range
- accumulate : sums up a range of elements
- inner_product : returns the inner product
- next_permutation :

```
int main()
{
    std::string s = "aba";
    std::sort(s.begin(), s.end());
    do {
        std::cout << s << '\n';
    } while(std::next_permutation(s.begin(), s.end()));
}
```

→

```
aab
aba
baa
```

homework

- write a simple function which returns the minimum of 2 integers
- write a simple function which returns the minimum of 2 doubles
- write a simple function which returns the minimum of 2 floats