Intro to the C++ Standard Template Library (STL)

- The STL is a collection of related software elements
  - Containers
    - Data structures: store values according to a specific organization
  - Iterators
    - Variables used to give flexible access to the values in a container
  - Algorithms
    - Functions that use iterators to access values in containers
    - Perform computations that modify values, or creates new ones
  - Function objects
    - Encapsulate a function as an object, use to modify an algorithm
- The STL makes use of most of what we've covered
  - Extensive use of function and class templates, concepts
- The STL makes use of several new ideas too
  - typedefs, traits, and associated types

Basic Requirements for an STL Container

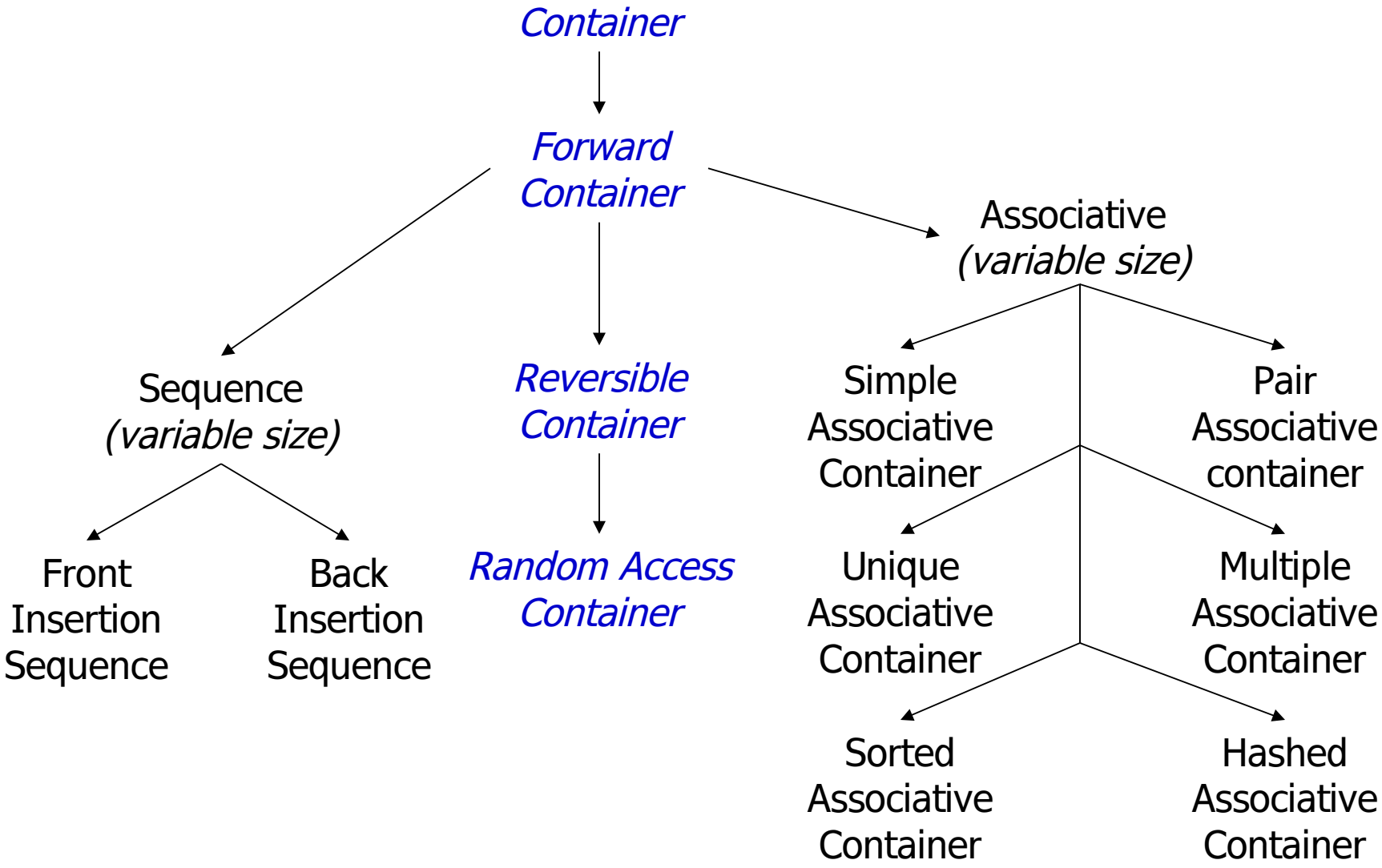
- Contains elements: *value semantics*
  - Containers may not overlap
  - An element belongs to at most one container.
    - may copy by value into other containers
    - object ownership can not be shared
  - Object's lifetime may not extend beyond that of the container.
    - object created no earlier than when container is constructed
    - contained object are destroyed when container is destroyed
  - Container may be fixed or variable size.
- Provide interfaces to contained values
  - Iterators with all elements contained in the range `[A.begin(),A.end())`
  - must define ancillary types: `value_type`, `pointer`, `const_pointer`, `reference`, `const_reference`, `difference_type` and `size_type`.
  - Should obey the *"principle of least surprise"*
    - For example a linked list would not provide **`[]`**
- Provide operations for a regular type
  - Assignable, Default Constructible, Equality Comparable

Classifying containers

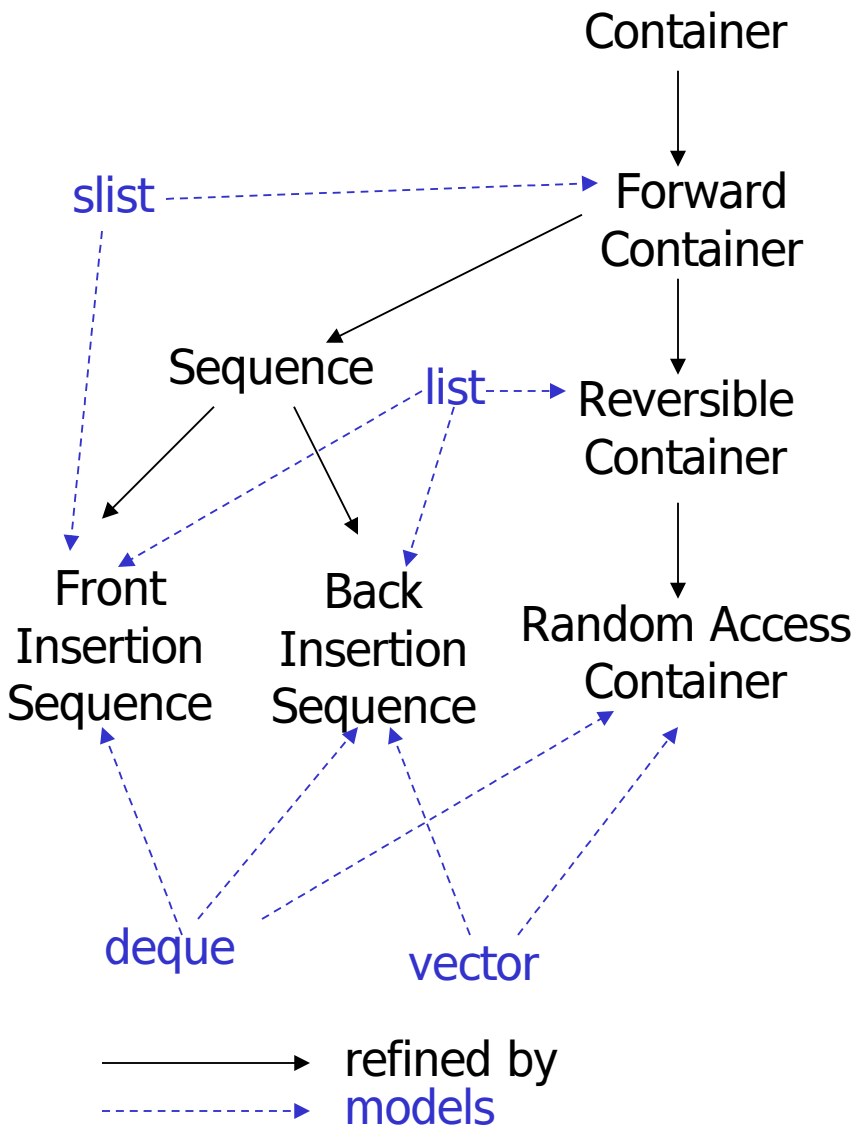
- Containers may be classified by the type of iterator:
  - Forward: supports forward iterators
  - Reversible: is a Forward Container whose iterators bidirectional iterators. A reversible container must define `reverse_iterator`, `const_reverse_iterator` and the methods `rbegin()` and `rend()`.
    - Reverse iterator range `[A.rbegin(),A,rend())`
  - Random Access: A reversible container whose iterators are random access iterators. It defines the operator `operator[]()`.

Hierarchy of STL Container Concepts

From: Matthew H. Austern, "Generic Programming and the STL"



General Container Concepts



- Notice containers can have multiple classifications
  - Useful to look at differences between data structures!
  - Back vs. front insertion
  - Forward vs. reversible vs. random access
- More general concepts higher in the hierarchy
- More specific concepts appear farther down

Container: Top of its Concept Hierarchy

Container

Invariants (for Container a):
- **valid range: [a.begin(), a.end()), but order is not guaranteed**
- **Range size: a.size() == distance(a.begin(), a.end())**
- **Completeness: Iterating through the range [a.begin(), a.end()) will access all elements.**

- Valid Expressions

	Complexity
Copy constructor `X(a)`	linear
Copy constructor `X b(a)`	linear
Assignment operator `b = a`	linear
Destructor `a.~X()`	linear
Beginning of range `a.begin()`	constant
End of range `a.end()`	constant
Size `a.size()`	linear -- O(N)
Maximum size `a.max_size()`	amortized, constant
Empty `a.empty()`	amoritized, constant
Swap `a.swap(b)`	linear – O(N)

Container

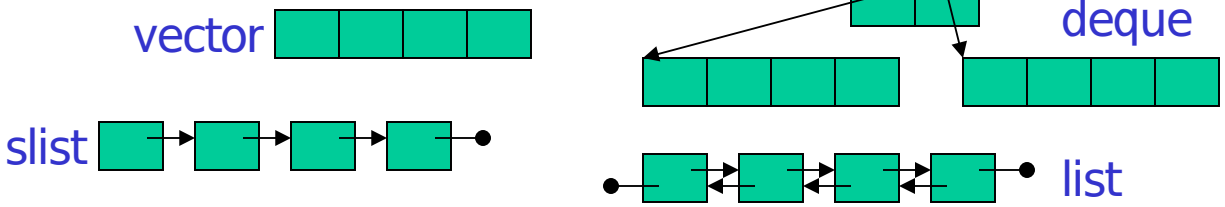
- Container concept:
  - Owns elements that it stores.
  - Provides methods to access elements
  - Defines an associated iterator type
  - Requires iterator to model the input_iterator concept
  - Elements are unordered.
  - Only one active iterator permitted.
  - Refinement of **Assignable**
- Associated types:
  - `value_type`: must model **Assignable**
  - `reference`: usually value_type&
  - `const_reference`: usually const value_type&
  - `pointer`: usually value_type*
  - `const_pointer`: usually const value_type*
  - `iterator`: must model input iterator. Expected its value, reference and pointer types are the same as the containers. Its not required to be mutable.
  - `const_iterator`: value type is expected to be the containers value type (not const value_type). Reference and pointer types expected to be const versions of containers.
  - `different_type`: signed integral type, represents distance between iterarors.
  - `size_type`: unsigned integral represents >0 value of difference type.

Forward Container

Container

↓

Forward Container

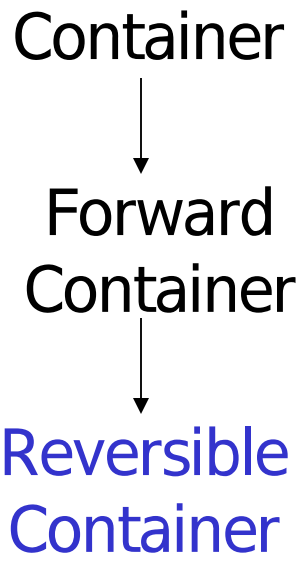


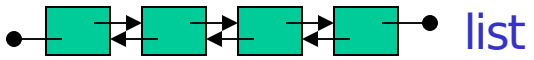
Refinement of Container, Equality Comparable, LessThan Comparable
equality semantics: sizes are equal and each element is equal
Invariants (for Forward Container a):
- Ordering: ordering is consistent across accesses, providing no mutations of occurred.

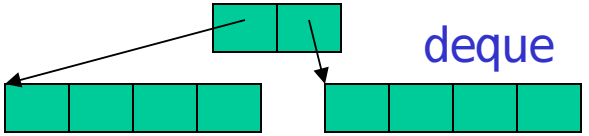
- Additional Expressions

	Complexity
– Equality **a == b**	linear
– Inequality **a != b**	linear
– Less than **a < b**	linear
– Greater than **a > b**	linear
– Less than or equal **a <= b**	linear
– Greater than or equal **a >= b**	linear

Reversible Container







Refinement of Forward Container whose iterators are bidirectional.
Introduces types: reverse_iterator and const_reverse_iterator
Invariants:
- Valid range: [a.rbegin(), a.rend())
- Equavalence of ranges: forward and reverse distance is the same.

- Additional Expressions
  - Beginning of reverse range **`a.rbegin()`** — Complexity: amortized constant
  - End of reverse range **`a.rend()`** — Complexity: amortized constant

Random Access Container

Container

↓

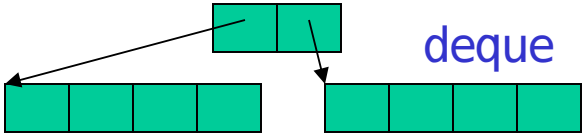
Forward Container

↓

Reversible Container

↓

Random Access Container



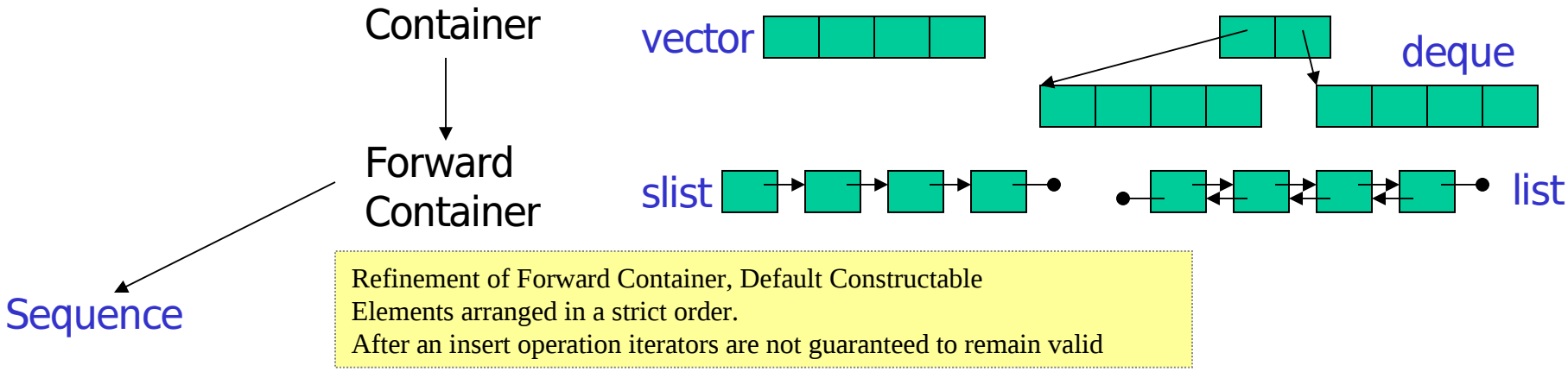
Refinement of Reversible Container whose iterator is Random Access.

- Additional Expressions
  - Element access **`a[n]`**

Complexity

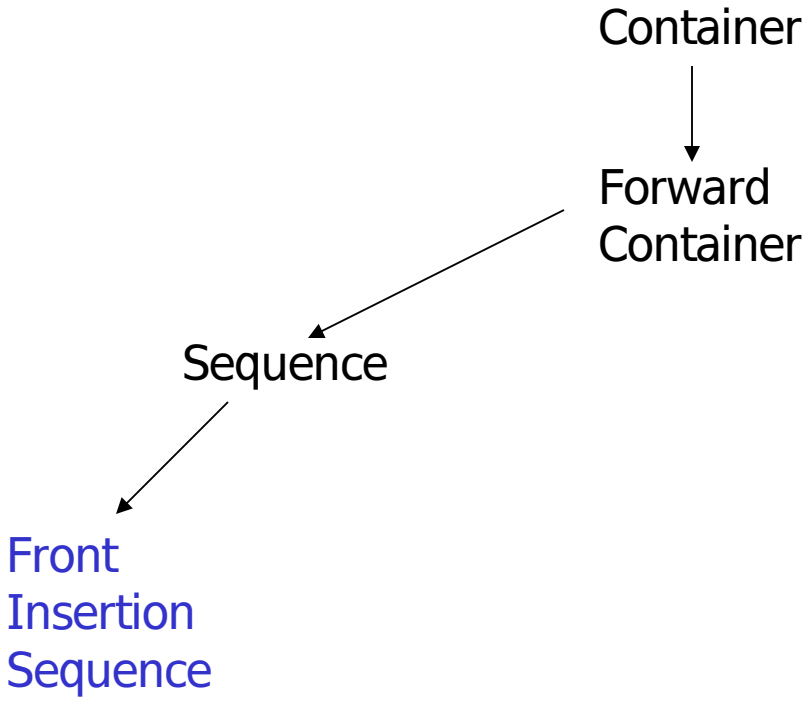
Amortized constant

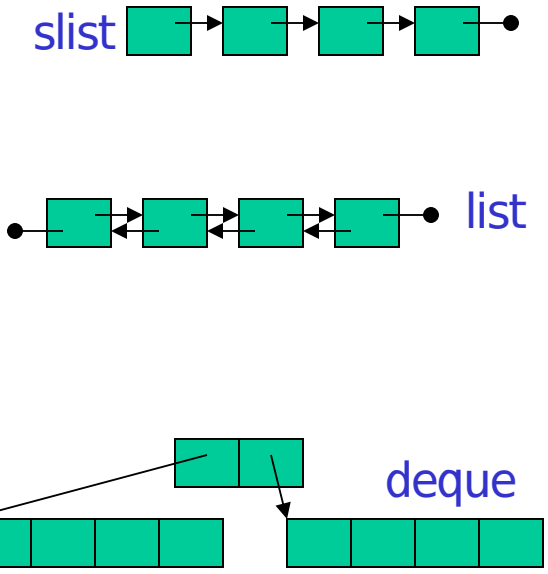
Sequence



- Additional Expressions — Complexity
  - Fill constructor `X(n,t)` — linear
  - Fill constructor `X(n)` `(same as X(n,T())` — linear
  - Default constructor `X()` `(same as X(0,T())` — linear
  - Range constructor `X(i,j)` — linear
  - Insert `a.insert(p,t)` — sequence-dependent
  - Fill insert `a.insert(p,n,t)` — linear
  - Range insert `a.insert(p,i,j)` — linear
  - Erase `a.erase(p)` — sequence-dependent
  - Range erase `a.erase(p,q)` — linear
  - Front `a.front()` — amortized constant

Front Insertion Sequence



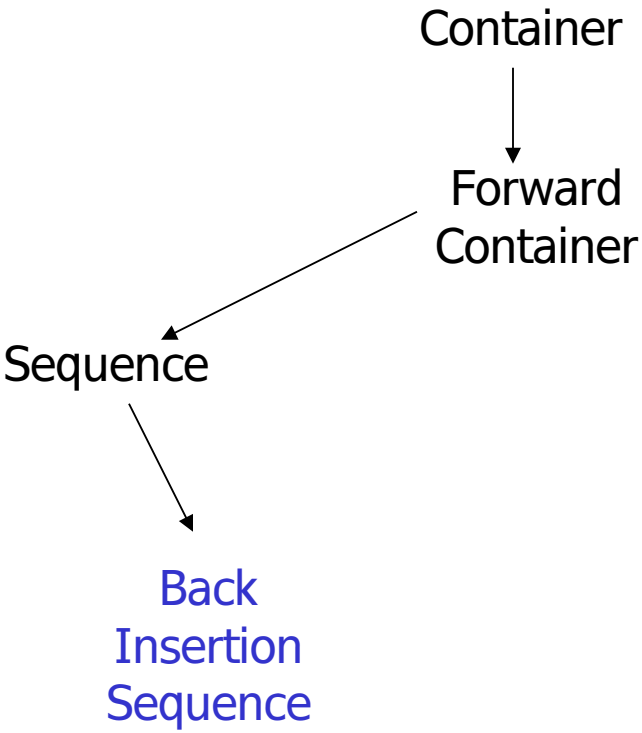


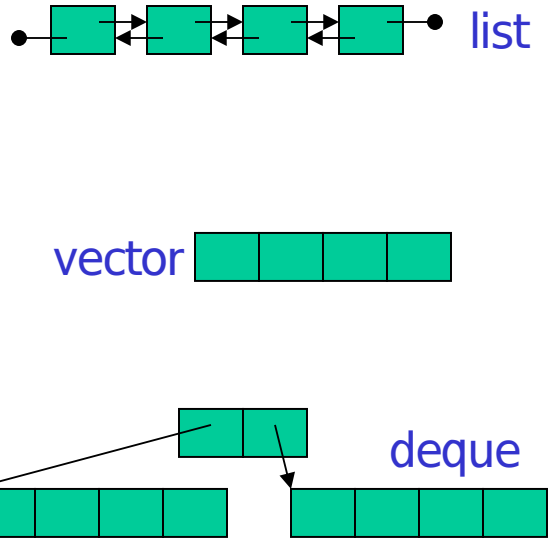
- Additional Expressions
  - a.front()
  - Push front **`a.push_front(t)`**
  - Pop front **`a.pop_front(t)`**

Complexity

constant
constant

Back Insertion Sequence





- Additional Expressions

	Complexity
Back **`a.back()`**	constant
Push back **`a.push_back(t)`**	constant
Pop back **`a.pop_back(t)`**	constant

Sequential Containers

- Sequential Containers
  - vector: fast random access
  - list: fast insertion/deletion
  - deque: double-ended queue
- Sequential Containers Adaptors
  - stack: last in/first out stack
  - queue: First in/First out queue
  - priority queue: priority management
- Element types must support assignment and copy.
- Only vector and deque support subscripting

Sequence s

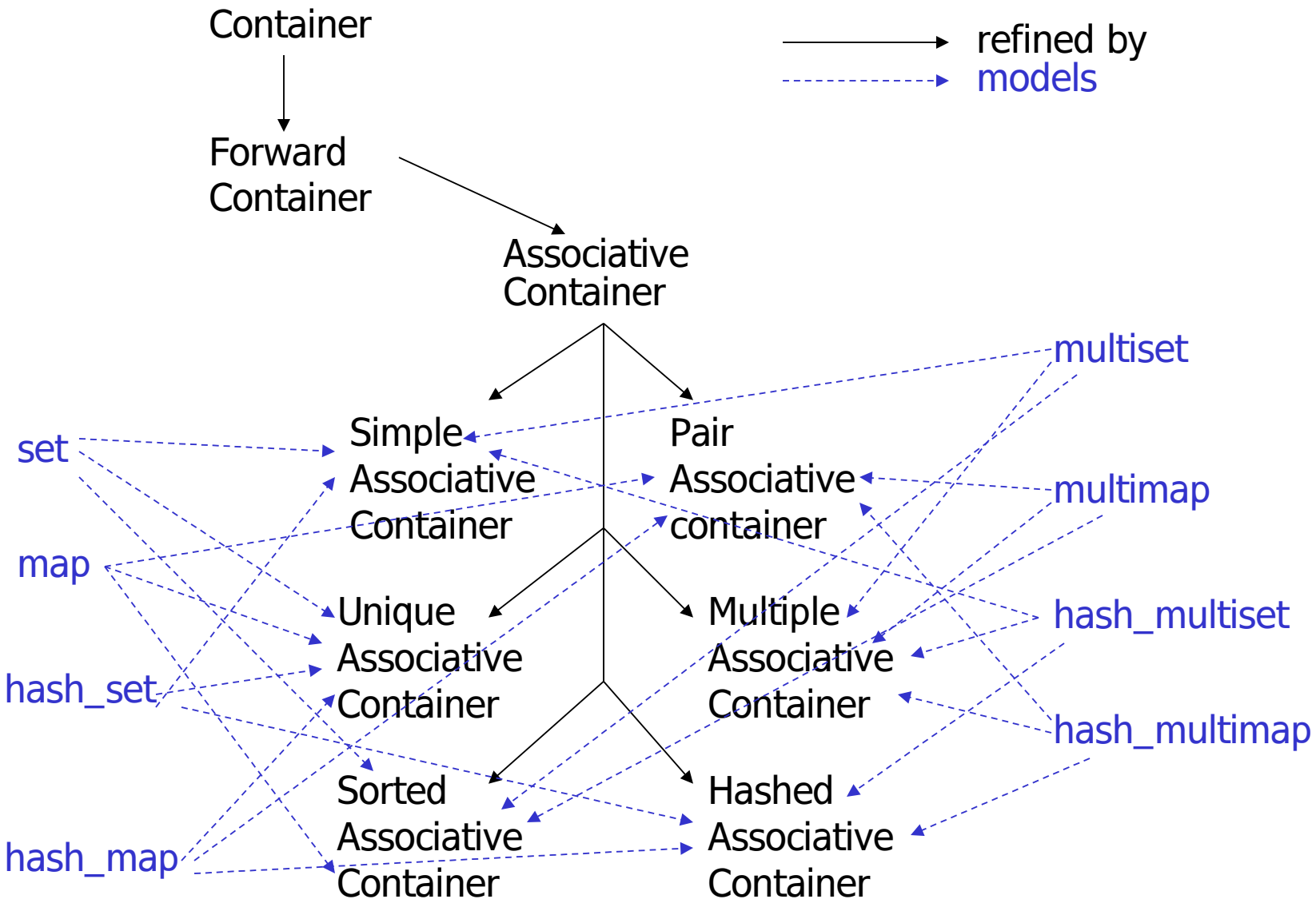
- vector can be used as a stack:
  - `push_back(), pop_back()`
  - `pop_back()` does not return a value, must use `back()`.
- List operations
  - `insert()`, `erase()` and `clear()`.
    - not as efficient on vectors.
    - a list container is optimized for inserting and removing from arbitrary locations within the sequence.
    - adding/removing elements may invalidate iterator

size and capacity

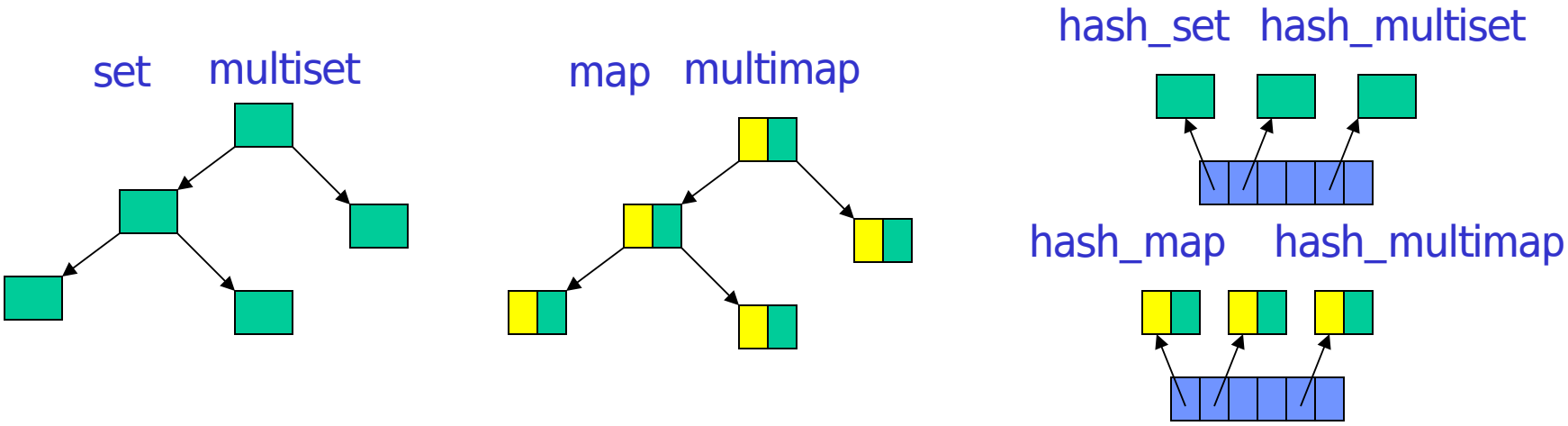
- the `size()` method returns the number of elements in the container
- the `capacity()` method indicates the maximum number of elements that can be stored in the current memory allocation – *vector only*.
  - `capacity()` – `size()` is the number of elments that can be added before memory must be reallocated.
- `max_size()` is the largest possible container of this type.
- calling `resize()` on a vector may move elements to another location invalidating any iterators.
- instantiating a container may result in a bad_alloc() exception

```
vector<int> v(10000);
```

Associative Container Concepts



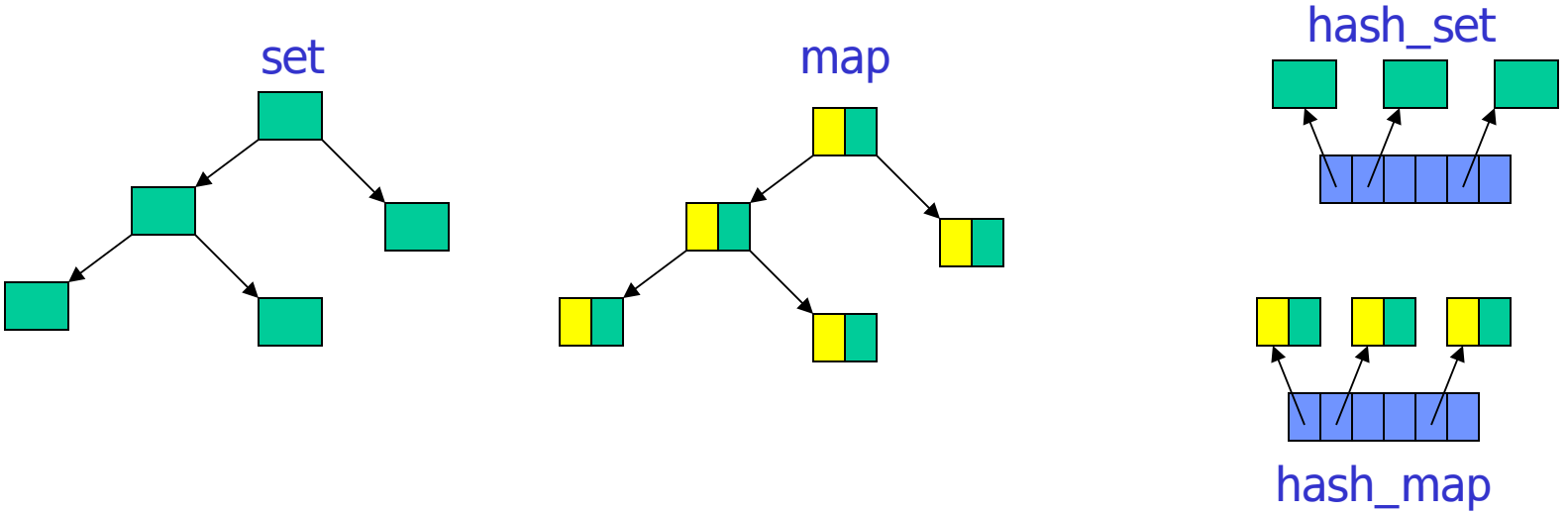
Associative Container



- Additional Expressions

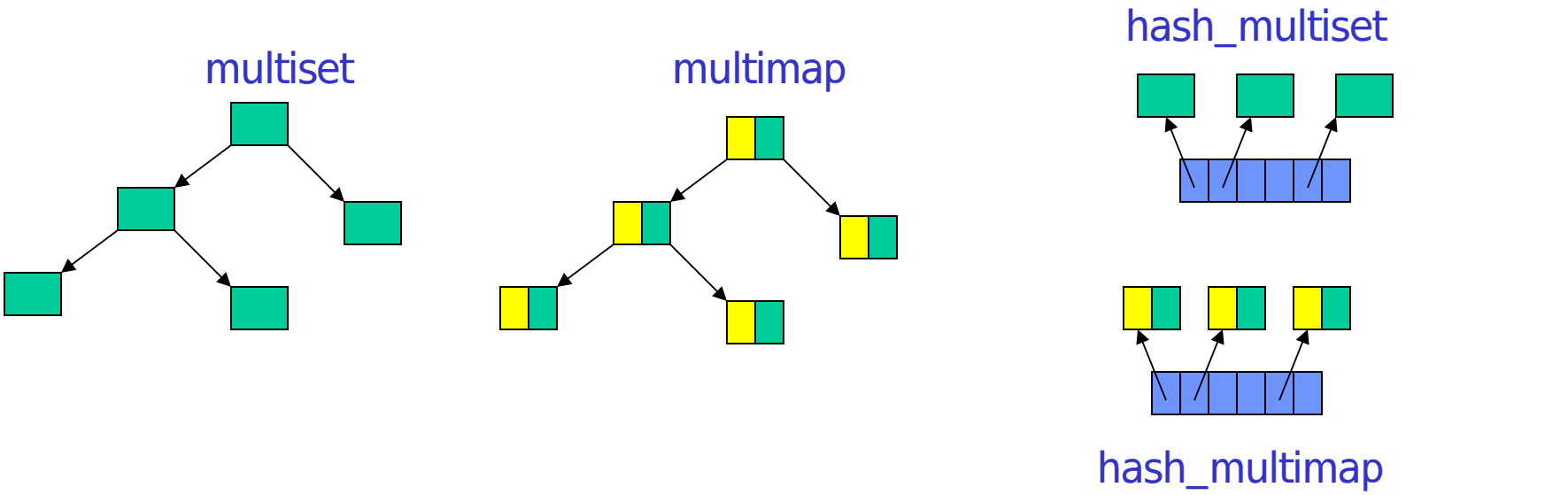
	Complexity
Default **`constructor X ()`**	constant
Default constructor **`X a;`**	constant
Find **`a.find(k)`**	logarithmic
Count **`a.count(k)`**	O(log(size())+count(k))
Equal range **`a.equal_range(k))`**	logarithmic
Erase key **`a.erase(k)`**	O(log(size())+count(k))
Erase element **`a.erase(p)`**	constant
Erase range **`a.erase(p,q)`**	O(log(size())+count(k))

Unique Associative Container



- Additional Expressions — Complexity
  - Range constructor `X a(i,j)` — linear
  - Insert element `a.insert(t)` — logarithmic
  - Insert range `a.insert(i,j)` — O(Nlog(size()+N))

Multiple Associative Container



- Additional Expressions
  - Range constructor **`X a(i,j)`**
  - Insert element **`a.insert(t)`**
  - Insert range **`a.insert(i,j)`**

Complexity

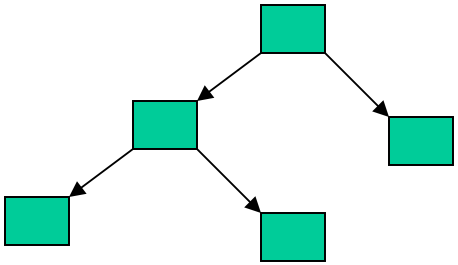
linear

logarithmic

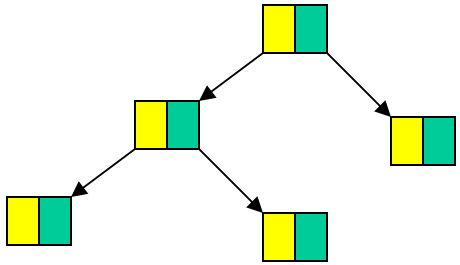
O(Nlog(size()+N))

Sorted Associative Container

set multiset



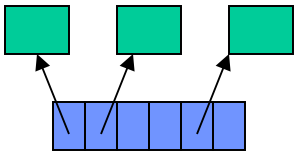
map multimap



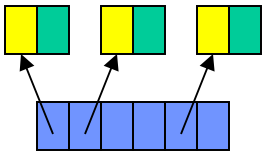
- Additional Expressions — Complexity
  - Default constructors `X (); X a;` — constant
  - Default constructor with comparator `X a(c)` — constant
  - Range constructor `X(i,j)` — O(NlogN)
  - Range constructor w/ comparator `X a(i,j,c)` — O(NlogN)
  - Key comparison `a.key_comp()` — constant
  - Value comparison `a.value_comp()` — constant
  - Lower bound `a.lower_bound(k)` — logarithmic
  - Upper bound `a.upper_bound(k)` — logarithmic
  - Equal range `a.equal_range(k)` — logarithmic
  - Insert with hint `a.insert(p,t)` — logarithmic

Hashed Associative Container

hash_set hash_multiset



hash_map hash_multimap



- Additional Expressions — Complexity
  - Default constructors `X (); X a;` — constant
  - Default constructor with bucket count `X a(n)` — constant
  - Default constructor with hash function `X a(n,h)` — constant
  - Default constructor with key equality `X a(n,h,k)` — constant
  - Range constructor `X a(i,j)` — linear
  - Range constructor with bucket count `X a(i,j,n)` — linear
  - Range constructor w/ hash fxn `X a(i,j,n,h)` — linear
  - Range constructor w/ key `eq X a(i,j,n,h,k)` — linear
  - Hash function `a.hash_funct()` — constant
  - Key equality `a.key_eq()` — constant
  - Bucket count `a.bucket_count()` — constant
  - Resize `a.resize()` — linear

Example Using map

- write a program that maps c-style strings for numbers specifying your own comparison operator (<)

```
// define a functor for comparing c-style strings
class CStringLess {
  public:
    bool operator()(const char *s1, const char *s2) {
      return strcmp(s1, s2) < 0;
    }
};
// define convenience typedefs
typedef map<const char *, const char *, CStringLess> mapType;
typedef mapType::value_type pairType;

mapType tbl; // the table
tbl["00"] = "Zero";  tbl["01"] = "One";  tbl["02"] = "Two";
tbl["03"] = "Three"; tbl["04"] = "Four"; tbl["05"] = "Five";
tbl["06"] = "Six";   tbl["07"] = "Seven";tbl["08"] = "Eight";
tbl["09"] = "Nine";
```

continued

- Looking for a value

```
mapType::const_iterator cit = tbl.find(“05”)
if (cit == tbl.end())
  … found it …
else
  … not found …
```

- Erase a value

```
mapType::iterator iter = tbl.find(eraseVal);
if (iter != tbl.end())
  tbl.erase(iter);
```

- Inserting values using `insert()`

```
pair<mapType::iterator, bool> ret =
        tbl.insert(make_pair(eraseVal, "XXXXX"));
if (ret.second)
  std::cout << "Inserted entry: “
            << ret.first->first << "\n";
```

Student Records

Num	LastName	FirstName	MName	Email	StdtId	DropCd	Score	Grade
1)	Alhaddad	Lorinda	Hang	al@cecX.wustl.edu	000007	EN	87	B
2)	Asnicar	Reynalda	Phebe	ar@cecX.wustl.edu	000000	EN	97	A
3)	Baudino	Ernesto	Rex	be@cecX.wustl.edu	000016	WD	-1	NG
4)	Bock	Ester	Jimmy	be@cecX.wustl.edu	000010	EN	88	B
5)	Bonavita	Elias	Johnathan	be@cecX.wustl.edu	000012	EN	71	C
6)	Botti	Maybell	Shawnta	bm@cecX.wustl.edu	000014	EN	27	F
7)	Dailey	Kyle	Quinn	dk@cecX.wustl.edu	000018	DP	-1	NG
8)	Debellis	Rusty	Gale	dr@cecX.wustl.edu	000009	EN	85	B
9)	Duldulao	Sherman	Orlando	ds@cecX.wustl.edu	000003	EN	95	A
10)	Hertweck	Carmelo	Garret	hc@cecX.wustl.edu	000011	EN	80	B
11)	Laughead	Troy	Kirby	lt@cecX.wustl.edu	000015	EN	39	F
12)	Lieuallen	Cristen	Erma	lc@cecX.wustl.edu	000001	EN	93	A
13)	Malsom	Anton	Darrell	ma@cecX.wustl.edu	000013	EN	72	C
14)	Mcbrayer	Jerald	Wendell	mj@cecX.wustl.edu	000019	DP	-1	NG
15)	Myer	Brandie	Aleen	mb@cecX.wustl.edu	000002	EN	92	A
16)	Schmid	Tarsha	Louis	st@cecX.wustl.edu	000008	EN	83	B
17)	Siroka	Odis	Tom	so@cecX.wustl.edu	000017	WD	-1	NG
18)	Tutaj	Keva	Venessa	tk@cecX.wustl.edu	000004	EN	88	B
19)	Ventrella	Jene	Reita	vj@cecX.wustl.edu	000005	EN	83	B
20)	Waz	Nereida	Sherill	wn@cecX.wustl.edu	000006	EN	85	B

Example using multimap

```
typedef vector<string> record_t;
typedef vector<record_t> roster_t;

roster_t roster;
map<string, record_t> nameMap;
multimap<string, record_t> gradeMap;

// Now get  roster records
while (getline(fin, line)) {
  vector<string> fields;
  // skip blank lines
  if (string2flds(fields, line, "\t", " \n\r") == 0)
    continue;
  // create student record, ignoring first field
  record_t rec(fields.begin()+1, fields.end());
  roster.push_back(rec);
```

continued

```
{ // … loop reading records
  // Add to name to roster mapping
  nameMap[fullName] = roster.back();
  // Add to grade to roster mapping
  gradeMap.insert(make_pair(rec[Grade], rec));
}

  // print the number of students receiving an A
  cout << "Students getting an A (" << gradeMap.count("A") << ")\n";

  // print names of students receivign an A
  multimap<string, record_t>::const_iterator iter =
                                gradeMap.lower_bound("A");
  for (; iter != gradeMap.upper_bound("A"); ++iter)
    cout << "\t" << iter->second[LastName]
         << ", " << iter->second[FirstName] << endl;

  // All students
  map<string, record_t>::const_iterator iterx = nameMap.begin();
  for (; iterx != nameMap.end(); ++iterx)
    cout << iterx->second[LastName] << endl;
```