Using Associative Containers

Lecture 13

Hartmut Kaiser

https://teaching.hkaiser.org/spring2024/csc3380/

Software Development Notes

Finite State Machine

- A finite state machine is a mathematical construct
- It is a type of state transition diagram
  - Others include cellular automaton, petri nets, and Turing machines
  - Entities change state upon the trigger of an event
- Common uses
  - Discrete Event Simulations
  - Asynchronous Programming
  - UI Navigation

State Machine Diagram

- A State Machine diagram is a control flow diagram
- Shows discrete behavior of a part of a designed system through finite state transitions
- How events change an object over its life

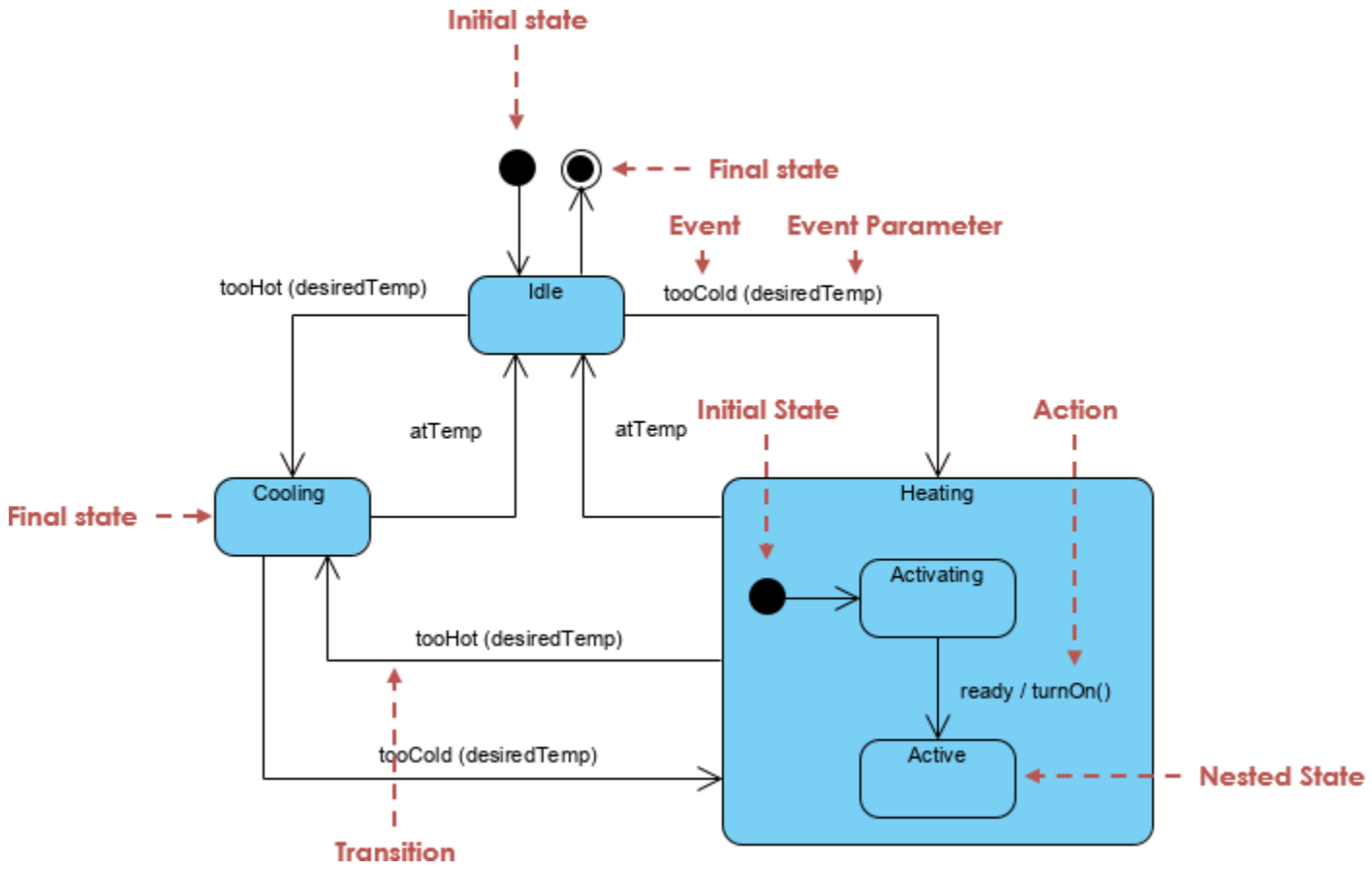
State Machine Diagram

- A **state machine** is a behavior that specifies the sequences of states an object goes through during its lifetime
  - In response to events, together with its responses to those events.
- A **state** is a condition or situation during the life of an object during which
  - It satisfies some condition
  - Performs some activity, or
  - Waits for some event.
- An **event** is the specification of a significant occurrence that has a location in time and space.
  - An event is an occurrence of a stimulus that can trigger a state transition.

State Machine Diagram

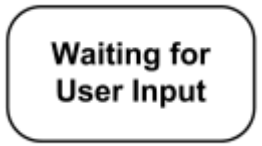
- A **guard condition** is evaluated after the trigger event for the transition occurs.
  - It is possible to have multiple transitions from the same source state and with the same event trigger, as long as the guard conditions don't overlap.
  - A guard condition is evaluated just once for the transition at the time the event occurs.
- A **transition** is a relationship between two states indicating that
  - An object in the first state will perform certain actions and
  - Enter the second state when a specified event occurs and specified conditions are satisfied.
- An **action** is an executable atomic computation that results in a change in the state of the model or the return of a value.

Finite State Machine

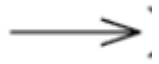


Key State Machine Elements

- Simple State: rounded rectangle

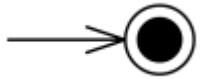


- State Transition, solid line with open arrow



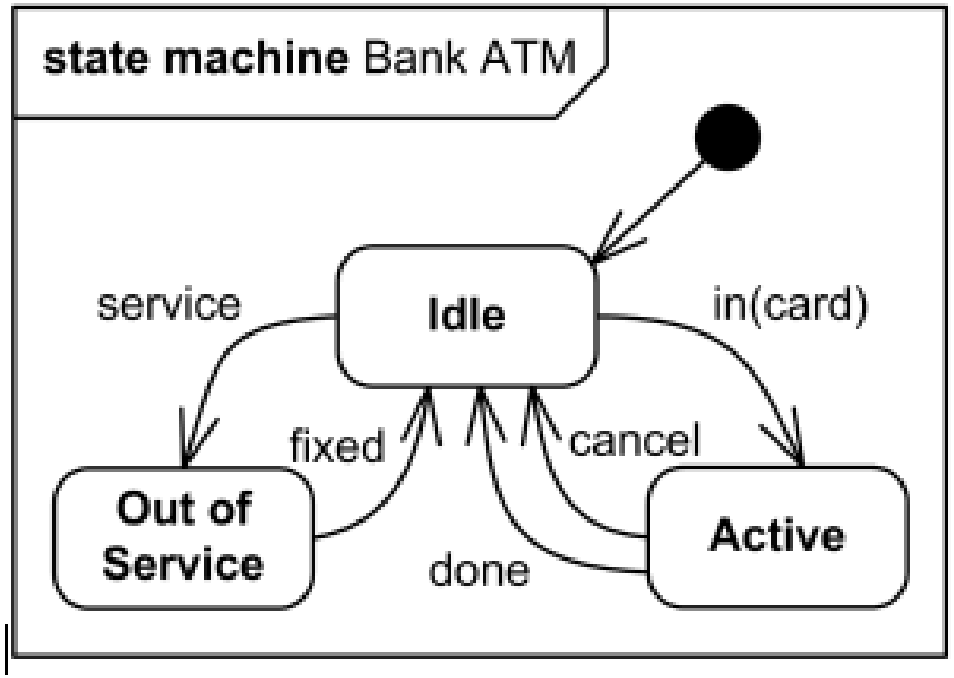
- Initial (Start) State, solid circle

- Final (End) State, solid circle surrounded by solid line



https://www.uml-diagrams.org/state-machine-diagrams.html

Example State Machine Diagram: Bank ATM



https://www.uml-diagrams.org/state-machine-diagrams.html

Associative Containers

Abstract

- Associative containers arrange their elements in a certain sequence based on an ordering criteria for the elements themselves
- They employ ordering to quicker find elements
- Often they store key/value pairs, ordering the values based on the key
- We will investigate these containers and use maps to write compact and efficient look-up-intensive programs.

Why Associative Containers?

- Finding an element in sequential containers entails sequential search
  - Potentially slow if container has many elements
- Alternative is to keep elements in sequential container in certain order
  - Devise a special search strategy, not easy
  - Potentially slow to insert as it might reorder elements
- Another alternative is to use associative containers

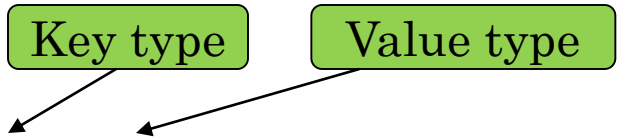
Associative Containers

- Associative containers automatically arrange their elements into a sequence that depends on the values of the elements themselves, rather than the sequence in which they were inserted
- Allows to locate element with particular value quickly
- The part which is used to locate an element is the *key*, which sometimes has an associated *value*

Counting Words

Counting Words

- Almost trivial with associative containers:




```
int main()
{
    std::string s;
    std::map<std::string, int> counters; // store each word and an
                                         // associated counter
    // read the input, keeping track of each word and
    // how often we see it
    while (std::cin >> s) {
        ++counters[s];
    }

    // write the words and associated counts
    for (std::map<std::string, int>::const_iterator it = counters.begin();
         it != counters.end(); ++it) {
        std::cout << it->first << "\t" << it->second << std::endl;
    }

    return 0;
}
```

Counting Words

- As `std::map` holds key/value pairs, we need to specify both: `std::map<string, int>`
  - Holds values of type `int` (the word counters) with a key of type `std::string` (the counted words)
  - We call this a 'map from `std::string` to `int`'
- 'Associative array', we use a string as the index (the *key*)
  - Very much like vectors, except that index can be any type, not just integers
- The entries are kept ordered based on
  - `operator<(Key, Key)` (total ordering of keys)
  - Explicit comparison function (weak strict ordering of keys):
    - `std::map<string, int, compare_func>`

Associative Container: `std::map`

- Usually implemented as a (self-)balanced red-black tree
  - Elements that compare less are left children, otherwise right children
- Tree stores `std::pair` of key/values

```
std::map<int, std::string> m = {
    { 42, "forty-two" },
    { 99, "ninety-nine" },
};
```

m

header.parent	A
header.left	A
header.right	B
size	2

↔

A

parent	m.header
left	null
right	B
color	black
data	42, "forty-two"

↔

B

parent	A
left	null
right	null
color	red
data	99, "ninety-nine"

- Complexities of operations
  - Insert, delete, find: `O(log N)`

Counting Words

- Necessary: `#include <map>`

```
while (std::cin >> s) ++counters[s];
```

- Indexing `operator[]`: invoked with string '`s`'
- Returns reference to integer value associated with string '`s`'
  - We increment this integer: counting words
  - If no entry representing string '`s`' exists, new entry is created and value initialized (integer is set to zero)

```
std::cout << it->first << "\t" << it->second << endl;
```

- Iterator '`it`' refers to both, key and value
  - `std::pair`: pair of arbitrary types, stored in map
  - The parts are named: `first`, `second`

Counting Words

- Almost trivial with associative containers:

```
int main()
{
    std::string s;
    std::map<std::string, int> counters; // store each word and an
                                         // associated counter
    // read the input, keeping track of each word and
    // how often we see it
    while (std::cin >> s) {
        ++counters[s];
    }

    // write the words and associated counts
    for (auto const& [key, value] : counters) {
        std::cout << key << "\t" << value << std::endl;
    }

    return 0;
}
```

An Input for the Word Counting Program

This lecture and the next presents the STL (the containers and algorithms part of the C++ standard library). It is an extensible framework dealing with data in a C++ program. First, I present the general ideal, then the fundamental concepts, and finally examples of containers and algorithms. The key notions of sequence and iterator used to tie containers (data) together with algorithms (processing) are presented. Function objects are used to parameterize algorithms with “policies”.

Output (Word Frequencies)

(data): 1
(processing): 1
(the: 1
C++: 2
First,: 1
Function: 1
Fundamental: 1
I: 1
It: 1
STL: 1
The: 1
This: 1
a: 1
algorithms: 3
algorithms.: 1
an: 1
and: 5
are: 2
concepts,: 1
containers: 3
data: 1
dealing: 1
examples: 1
extensible: 1
finally: 1
framework: 1
general: 1
ideal,: 1
in: 1
is: 1
iterator: 1
key: 1
lecture: 1
library).: 1
next: 1
notions: 1
objects: 1
of: 3
parameterize: 1
part: 1
present: 1
presented.: 1
presents: 1
program.: 1
sequence: 1
standard: 1
the: 5
then: 1
tie: 1
to: 2
…

Other Associative Containers

- `std::multimap<K, V>`: Same as map, however doesn't enforce uniqueness of keys
- `std::set<T>`: Same as map, except no 'values', just 'keys'
- `std::multiset<T>`: Same as set, however doesn't enforce uniqueness of keys
- Starting C++23 (gcc/clang option `--std=c++23`, msvc option `/std:c++23`)
  - `std::flat_map`, `std::flat_set`, `std::flat_multimap`, `std::flat_multiset`
  - Same interface as containers above, just implemented on top of `std::vector`
  - Re-arrange (sort) elements after each insertion

Splitting a Line into Words

Splitting a Line into Words

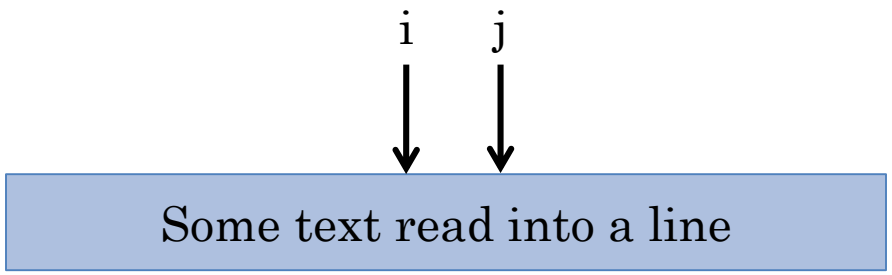
- We'll write a function which takes a whole line of input and returns a vector of strings holding the single words of that line:

```
std::vector<std::string> split(std::string const& s);
```

- Strings support indexing in the same way as vectors:
  - `s[0]`: refers to the first character in the string 's'
  - `s[s.size()-1]`: refers to the last character in a string
- Our function will find indices 'i' and 'j' delimiting each of the words, where the range of characters `[i, j)` constitutes the word

Splitting a Line into Words

- This looks like:



- Words are split at whitespace characters
  - Very similar to the processing during stream input into a string

Splitting a Line into Words

```
std::vector<std::string> split(std::string const& s)
{
    std::vector<std::string> words;
    std::vector<std::string>::size_type i = 0;

    // invariant: we have processed characters [original value of i, i)
    while (i != s.size()) {
        // ignore leading blanks, find begin of word

        // find end of next word

        // if we found some non-whitespace characters, store the word
    }
    return words;
}
```

Splitting a Line into Words

- Ignore leading blanks

```
// invariant: characters in range [original i, current i)
// are all spaces
while (i != s.size() && std::isspace(s[i]))   // short-circuiting
    ++i;
```

- Find end of next word

```
// find end of next word
auto j = i;

// invariant: none of the characters in range
// [original j, current j) is a space
while (j != s.size() && !std::isspace(s[j])) // short-circuiting
    ++j;
```

Splitting a Line into Words

- Store the word if any

```
// if we found some non-whitespace characters
if (i != j) {
    // copy from s starting at i and having j – i characters
    words.push_back(s.substr(i, j - i));
    i = j;
}
```

Splitting a Line into Words

```
std::vector<std::string> split(std::string const& s)
{
    std::vector<std::string> words;
    std::vector<std::string>::size_type i = 0;

    while (i != s.size()) {
        while (i != s.size() && std::isspace(s[i]))  ++i;

        auto j = i;
        while (j != s.size() && !std::isspace(s[j]))  ++j;

        if (i != j) {
            words.push_back(s.substr(i, j - i));
            i = j;
        }
    }
    return words;
}
```

Splitting a Line into Words: Simplified

```
std::vector<std::string> split(std::string const& s)
{
    std::stringstream str = s;

    std::vector<std::string> words;
    std::string word;

    while (str >> word) {
        words.push_back(word);
    }

    return words;
}
```

Generating a Cross-Reference

Generating a Cross-Reference Table

- Write a program to generate a cross-reference table that indicates where (what line) each word occurs in the input
  - Read a line at a time, allowing to associate line numbers with words
  - Split line into words
  - Store more data in a map: all lines a particular word occurred on

```
std::map<std::string, std::vector<int>>
```

Generating a Cross-Reference Table

- Find all the lines that refer to each word in the input:

```
std::map<std::string, std::vector<int>>
xref(std::istream& in,
     std::vector<std::string> find_words(std::string const&) = split)
{
    std::string line;                                  // current line
    int line_number = 0;                               // current line number
    std::map<std::string, std::vector<int>> ret;       // cross reference table

    // read the next line
    while (std::getline(in, line)) {
        // store current line number for each word
        // ...
    }
    return ret;
}
```

Generating a Cross-Reference Table

- Default argument specification:

```
std::map<std::string, std::vector<int>>
xref(std::istream& in,
     std::vector<std::string> find_words(std::string const &) = split);
```

- Allows to leave out this argument at invocation:

```
// uses split() to find words in the input stream
... = xref(std::cin);

// uses the function named find_urls to find words
... = xref(std::cin, find_urls);
```

Generating a Cross-Reference Table

- Store current line number for each word:

```
while (std::getline(in, line)) {
    // adjust current line number
    ++line_number;

    // break the input line into words
    std::vector<std::string> words = find_words(line);

    // remember that each word occurs on the current line
    for (auto const& s: words)
    {
        ret[s].push_back(line_number);   // see next slide
    }
}
```

Generating a Cross-Reference Table

- What is this doing here:

```
ret[s].push_back(line_number);
```

  - `s`: the current word
  - `ret[s]`: returns a reference to the value associated with the key ‘`s`’ yielding the vector of line numbers
    - If this is the first occurrence, an empty vector is put into the map
  - `ret[s].push_back()`: adds the current line number to the end of the vector

Printing the Cross-Reference

- Print the generated map:

```
int main() {
    // call xref using split by default
    std::map<std::string, std::vector<int>> xrefmap = xref(std::cin);

    // write the results
    for (auto it = xrefmap.begin(); it != xrefmap.end(); ++it)
    {
        // write the word followed by one or more line numbers
        std::cout << it->first << " occurs on line(s): ";

        auto line_it = it->second.begin();
        std::cout << *line_it++;    // write the first line number

        // write the rest of the line numbers, if any
        std::for_each(line_it, it->second.end(), [](int line) { std::cout << ", " << line; });
        std::cout << endl;          // write a new line to separate each word from the next
    }
    return 0;
}
```

Printing the Cross-Reference

```
Whether I shall turn out to be the hero of my own life
or whether that station will be held by anybody else
these pages must show
^Z

I occurs on line(s): 1
Whether occurs on line(s): 1
anybody occurs on line(s): 2
be occurs on line(s): 1, 2
by occurs on line(s): 2
else occurs on line(s): 2
held occurs on line(s): 2
hero occurs on line(s): 1
...
```

Generating Sentences

Generating Sentences

Categories	Rules
<noun>	cat
<noun>	dog
<noun>	table
<noun-phrase>	<noun>
<noun-phrase>	<adjective> <noun-phrase>
<adjective>	large
<adjective>	brown
<adjective>	absurd
<verb>	jumps
<verb>	sits
<location>	on the stairs
<location>	under the sky
<location>	wherever it wants
<sentence>	the <noun-phrase> <verb> <location>

- Example: `the table jumps wherever it wants`

Representing the Rules

- Categories, rules, and 'normal' words
  - Categories: enclosed in angle brackets
  - Right hand side is a rule consisting out of a sequence of categories and words
- How to represent/store categories?
  - Let's use a `std::map` to associate the categories with the corresponding rules
  - Several rules for same category

```
using category = std::string;
using rule = std::vector<std::string>;
using rule_collection = std::vector<rule>;
using grammar = std::map<category, rule_collection>;
```

Reading the Grammar

```
// read a grammar from a given input stream
grammar read_grammar(std::istream& in)
{
    grammar ret;
    std::string line;
    // read the input
    while (std::getline(in, line)) {
        // split the input into words
        std::vector<std::string> entry = split(line);
        if (!entry.empty()) {
            // use the category to store the associated rule
            ret[entry[0]].push_back(
                rule(entry.begin() + 1, entry.end()));
        }
    }
    return ret;
}
```

Generating the random Sentence

- Start off with category `<sentence>`
  - Assemble the output in pieces from various rules
  - Result is a `std::vector<std::string>` holding the words of the generated sentence

```
std::vector<std::string> generate_sentence(grammar const& g)
{
    std::vector<std::string> ret;
    generate(g, "<sentence>", ret);
    return ret;
}
```

Generating the random Sentence

- Our algorithm generate() knows how to query the grammar and how to collect the words
- Needs to decide whether a string is a category:

```
// return true if 's' represents a category
bool bracketed(std::string const& s)
{
    return s.size() > 1 && s[0] == '<' && s[s.size() - 1] == '>';
}
```

- If it's a category, look up rule and expand it
- If it's not a category, copy word to output

Generating the random Sentence

```
void generate(grammar const& g, std::string const& word, std::vector<std::string>& ret)
{
    if (!bracketed(word)) {
        ret.push_back(word);
    }
    else {
        // locate the rule that corresponds to word
        auto it = g.find(word);
        if (it == g.end()) throw std::logic_error("empty rule");

        rule_collection const& c = it->second;  // fetch the set of possible rules

        rule const& r = c[nrand(c.size())];     // from which we select one at random

        // recursively expand the selected rule
        for (auto it = r.begin(); it != r.end(); ++it)
            generate(g, *i, ret);
    }
}
```

Selecting a Random Element

```
#include <random>

// return a random integer in the range [0, n)
int nrand(int n)
{
    static std::mt19937 mt;    // Mersenne Twister engine

    std::uniform_int_distribution<int> dist(0, n - 1);
    return dist(mt);
}
```

Function local statics

- A function-local static variable
  - Is initialized once whenever the function is called first
  - Stays ‘alive’ even after the function execution has finished
  - Retains it’s state between function calls
  - Is visible only from inside the function

Pulling everything together

```
int main() {
    // generate the sentence
    std::vector<std::string> sentence = generate_sentence(read_grammar(std::cin));

    // write the first word, if any
    auto it = sentence.begin();
    if (!sentence.empty())
        std::cout << *it++;

    // write the rest of the words, each preceded by a space
    std::for_each(it, sentence.end(), [](std::string const& s) {
        std::cout << " " << s;
    });

    std::cout << std::endl;
    return 0;
}
```

Pulling everything together

```
int main() {
    // generate the sentence
    std::vector<std::string> sentence = generate_sentence(read_grammar(std::cin));

    // write the words, separated by a space
    bool first = true;
    for(auto const& s : sentence) {
        if (!first) {
            std::cout << " ";
            first = false;
        }
        std::cout << s;
    }

    std::cout << std::endl;
    return 0;
}
```

Selecting a Random Element

- Why does it print the same sentence whenever run?
  - Random number generators are not random
  - Generate a sequence of numbers with certain statistical properties
  - Each time they are used they generate the same sequence of numbers (by default)
- Each generator can be initialized using a 'seed' causing it to generate different sequences of numbers
  - Let's use a truly random number as the seed

Selecting a Random Element

```
#include <random>

// return a random integer in the range [0, n)
int nrand(int n)
{
    static std::random_device rd;    // truly random number generator
    static std::mt19937 mt(rd());    // seeded Mersenne Twister engine

    std::uniform_int_distribution<int> dist(0, n - 1);
    return dist(mt);
}
```

Performance Considerations

- Unlike to associative containers in other languages, `std::map` is not implemented as a hash table
  - For each key type those need a hash function
  - Performance is exquisitely sensitive to the details of this hash function.
  - There is usually no easy way to retrieve the elements of a hash table in a useful order.
- C++ associative containers are hard to implement in terms of hash tables:
  - The key type needs only the < operator or equivalent comparison function
  - Associative-container elements are always kept sorted by key
- C++ has `std::unordered_map<>`, `std::unordered_set<>`, etc.
  - Those are hash tables

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP