Search Algorithms I

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, September 16, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ Expressions
- ✓ Parameter passing I
- ✓ Operator overloading
- ✓ Parameter passing II
- ✓ Invisible functions I
- ✓ Integer types
- ✓ Managing resources in a class
- ✓ Containers & iterators
- ✓ Invisible functions II
- ✓ Error handling
- ✓ Using exceptions
- ✓ A little about Linked Lists

Major Topics: S.E. Concepts

- ✓ Invariants
- ✓ Testing
- ✓ Abstraction

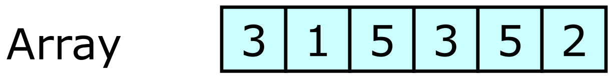
DONE

Review

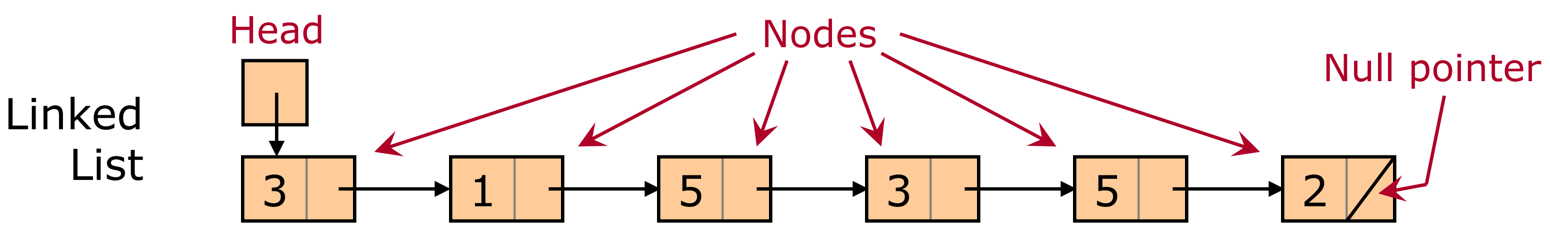
Review
A Little about Linked Lists [1/2]

We looked briefly at a container called a **Linked List**.

- Like an array, a Linked List stores a sequence of data items.



- A Linked List is made of **nodes**. Each has a single data item and a pointer to the next node, or a null pointer at the end of the list.

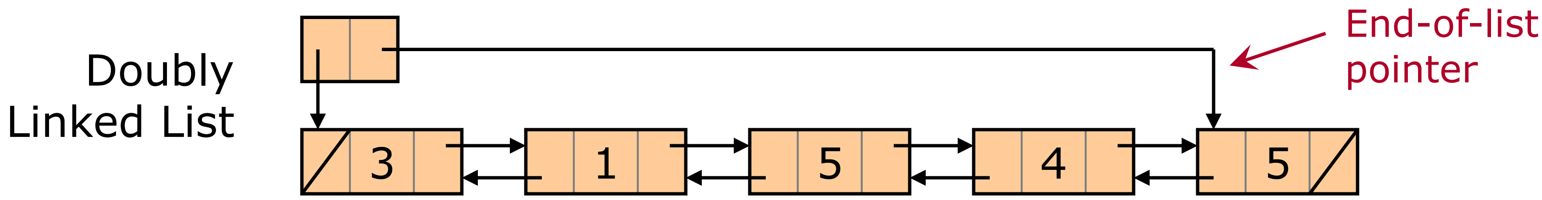


- These pointers are the only way to find the next item. A Linked List is a one-way sequential-access structure.

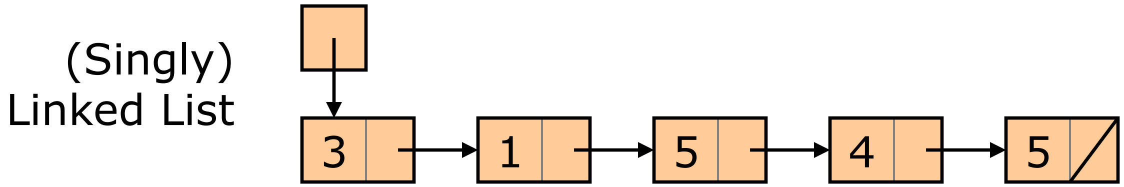
See `llnode.h` *for a definition of a Linked List node.*
See `list_size.cpp` *for a program that uses this node.*

Review
A Little about Linked Lists [2/2]

In a **Doubly Linked List**, each node has two pointers: next-node (null at the end) and previous-node (null at the beginning).



To make it clear what we are talking about, the one-pointer-per-node Linked List has a longer name: **Singly Linked List**.



Unit Overview
Recursion & Searching

Major Topics

- ✓ Introduction to recursion
- Search algorithms I
- Recursion vs. iteration
- Search algorithms II
- Eliminating recursion
- Search in the C++ STL
- Recursive backtracking

Review
Introduction to Recursion [1/2]

A **recursive** algorithm is one that makes use of itself.

- An algorithm solves a problem. If we can write the solution of a problem in terms of the solutions to **smaller** problems of the same kind, then recursion may be called for.
- There must be a smallest problem, which we solve directly. This is a **base case**. (Others are **recursive cases**.)

Similarly, a **recursive** function is one that calls itself.

```
int mult(int a, int b)
{
    if (a <= 1)
        return a == 1 ? b : 0;

    int ax = (a >> 1);
    int m1 = mult(ax, b);
    return m1 + m2(a, ax, b);
}

int m2(int a, int ax, int b)
{
    return mult(a-ax, b);
}
```

Base case (the `if (a <= 1) return a == 1 ? b : 0;` part)

Recursive case (the remaining three lines of `mult`)

Direct recursion (`mult(ax, b)`)

Indirect recursion (`m2(a, ax, b)`)

Introduction to Recursion [2/2]

The **Fibonacci numbers** (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, …) can be defined by the following **recurrence relation** with **initial conditions**:

- $F_0 = 0$.
- $F_1 = 1$.
- For $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$.

Based on this, we wrote a recursive function `fibo` that computes Fibonacci numbers.

See `fibo_first.cpp`*.*

Function `fibo` turned out to be extremely slow for anything other than small parameters. But do not conclude that recursion is slow! We will revisit `fibo`, rewriting it in various ways—including fast recursive versions.

Search Algorithms I

Binary Search — Description [1/3]

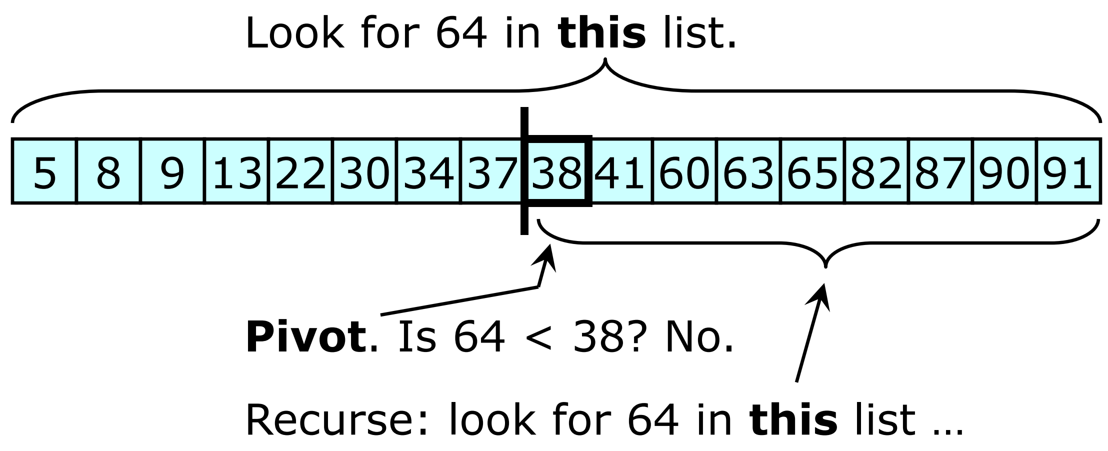
The **Binary Search** algorithm finds a given **key** in a **sorted list**.

- Here, *key* = thing to search for. Often there is associated data.
- In computing, **sorted** means in (some specified) order.

Procedure

- Pick an item in the middle of the list: the **pivot**.
- Compare the given key with the pivot.
- Using this, narrow search to top or bottom half of list. Recurse.

Example: Use Binary Search to search for 64 in the following list.



Binary Search — Description [2/3]

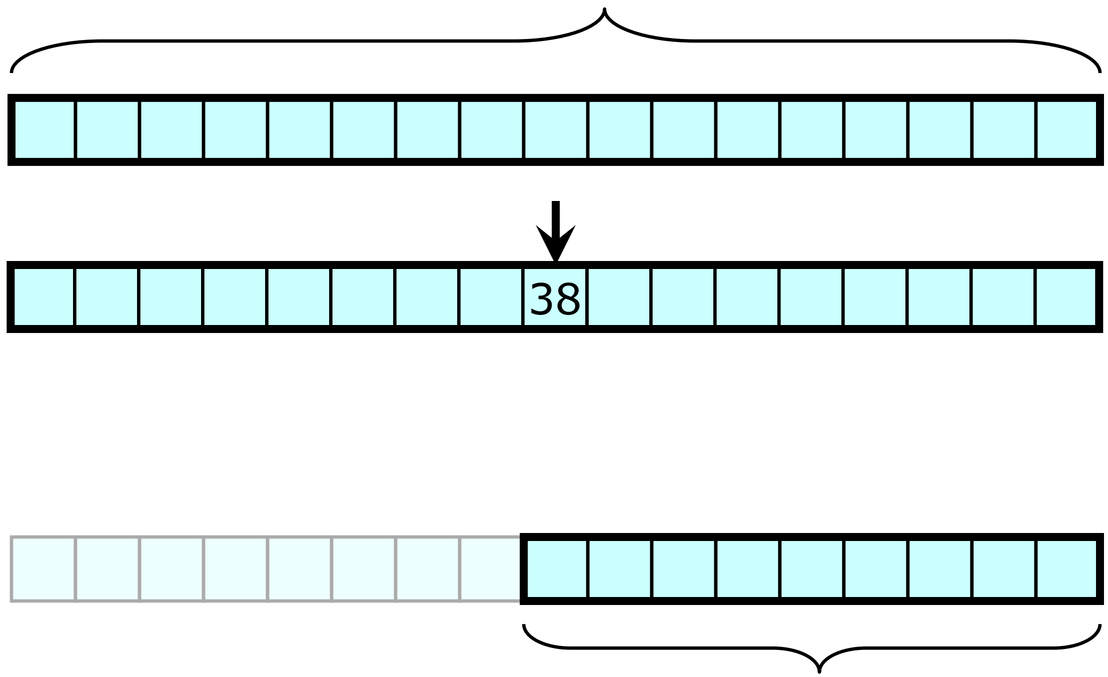
Remember that the algorithm does not see the value of any item in the list until it specifically retrieves that item.

So we might think of Binary Search as follows:

All values unknown

Look for 64 in this list:

Choose the pivot item, and get its value:



Is 64 < 38? No.

Recurse—look for 64 in this list:

Etc.

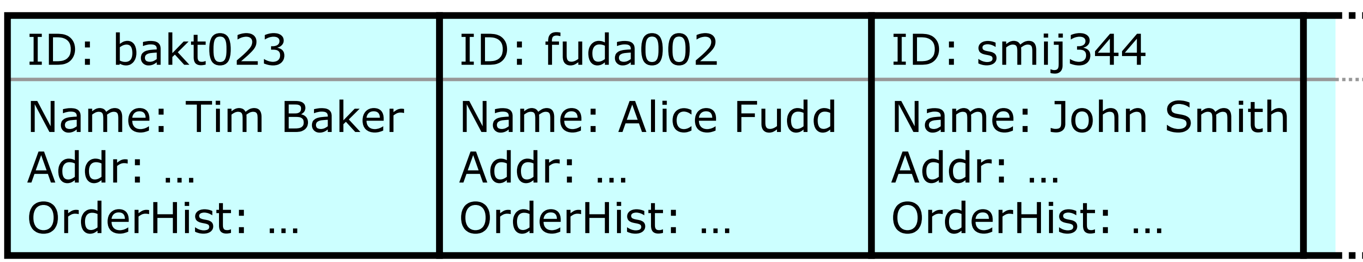
38 is not passed to the recursive call except as part of the list. All values unknown once again.

Binary Search — Description [3/3]

In most of my illustrations, the keys are integers. However:

- Keys for Binary Search can be almost anything that can be sorted.
- In practice, the key is often not the entire data item.

For example, consider a list of the customers of some business. Each customer is identified by a unique string, which is used as the key. The data item for each customer also includes the customer's name, address, and order history.



Since the keys are sorted, we can do Binary Search to find a particular key.

> This idea will also apply to other algorithms and data structures we cover.

Binary Search — Four Questions

1. How can we solve the problem using solutions to one or more smaller problems of the same kind?
   - Choose a pivot in the middle of the list. Compare, and then apply Binary Search to the top or bottom half of the list, as appropriate.

2. How much does each recursive call reduce the size of the problem?
   - It cuts the size of the list roughly in half.

3. What instances of the problem can serve as base cases?
   - List size is 0 or 1.

4. As the problem size shrinks, will a base case always be reached?
   - Yes—the size of a list cannot be negative.

Binary Search — Design Decisions

Let's write a function to do Binary Search.

How should the list to search be given?

- Parameters: two iterators specifying a range as usual—one pointing to the first item and one pointing just past the last item.

Should there be any other parameters?

- Yes, the key to search for.

The parameter types may vary. We will write a function template.

What should it return?

- There are several options:
  - Return a `bool`, indicating found or not found.
  - Return an iterator to the value found.
  - Return an iterator to the first equivalent value in the list.
  - Return two iterators specifying the range of equivalent values.
- Use the first option: return a `bool`. (So our function will not indicate *where* a key is in the list; we certainly *could* write such a function.)

Binary Search — CODE

TO DO

- Write a function template `binSearch` that does Binary Search, as designed on the previous slide.

Done. See `binsearch1.cpp`*.*

Better Binary Search — Equality vs. Equivalence

Let’s improve function template `binSearch`.

Here are some ideas.

Can we get away with using *nothing* that deals with the value type other than `operator<`?

In particular, if we use `operator<` to search for something, then we prefer to use `operator<` to check whether we have found it.

- **Equality**: `a == b`
- **Equivalence**: `!(a < b) && !(b < a)`

Using equivalence instead of equality lets us handle types that:

- Do not have `operator==`.
- Have `operator==`, but do not define it in a way that is consistent with `operator<`.

Improvement: Check equivalence in the base case. Never use “==” on the value type.

Search Algorithms I
Better Binary Search — Extra Computations

`binSearch` finds the size of the range (`last - first`) when finding the middle. The size is also used in the base case.

Improvement: Compute the size of the range once. Save this for later use.

`binSearch` does two checks to see if it is in a base case. Most of the time, these will both be executed.

Improvement: Only check for the base case once.

- This will make the base case more complicated. But the base case only happens once in a search; there may be many recursive calls.

`binSearch` takes the key by value. This requires copy construction, as well as destruction at the end of the function. These may be time-consuming if the keys are objects.

Improvement: Pass the key by reference-to-const.

Better Binary Search — More General Iterators

Random-access iterators can do pointer-style arithmetic:

- Adding Integers
  - `iter1 = iter2 + 3;`
  - `iter1 += 3;`
- Difference
  - `n = iter1 - iter2;`

In general, iterators may not support all of these operations.

However, we can get the same *results* for more general **forward iterators** with `std::advance` & `std::distance` (`<iterator>`).

- `std::advance(iter, n)` is like `iter += n`.
- `std::distance(iter1, iter2)` is like `iter2 - iter1`.
- These two functions are fast for random-access iterators; they may be slower for other iterators.

Improvement: Replace pointer arithmetic with `std::advance` & `std::distance`. Allow the parameters to be forward iterators.

Is this really an improvement? Maybe. Regardless, `std::advance` and `std::distance` are worth learning about.

Search Algorithms I
Better Binary Search — CODE

TO DO

- Improve function template `binSearch`:
  - Check equivalence in the base case. Never use “==” on the value type.
  - Compute the size of the range once. Save this for later use.
  - Only check for the base case once.
  - Pass the key by reference-to-const.
  - Replace pointer arithmetic with `std::advance` & `std::distance`. Allow the parameters to be forward iterators.

Done. See `binsearch2.cpp`*.*

Benefits

- Redundant computations are avoided.
- `binSearch` works with datasets that support a very limited set of operations. Only `operator<` is used on the value type (no copy ctor, dctor, or `operator==` required). Iterators can be forward iterators.
- The recursive and base cases work consistently, using `operator<`.

We are not done improving `binSearch` yet!