

Search Algorithms I

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, September 18, 2024

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

© 2005–2024 Glenn G. Chappell

Some material contributed by Chris Hartman

Unit Overview

Recursion & Searching

Topics

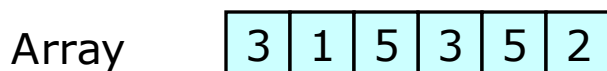
- ✓ ■ Arrays & Linked Lists
- ✓ ■ Introduction to recursion
 - Search algorithms I
 - Recursion vs. iteration
 - Search algorithms II
 - Eliminating recursion
 - Search in the C++ STL
 - Recursive backtracking

Review

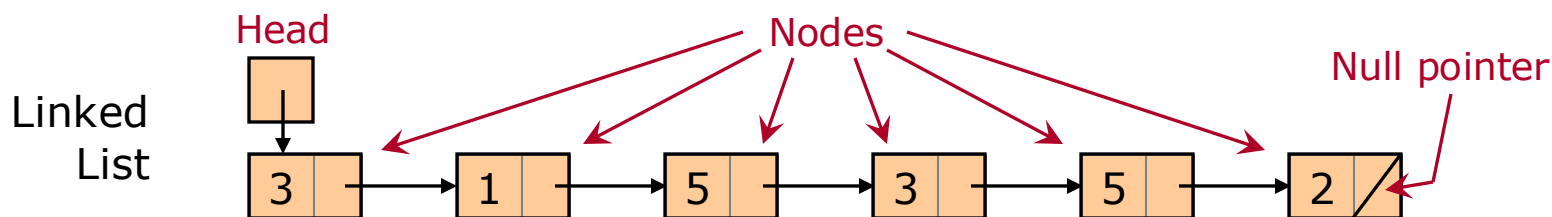
Review

Arrays & Linked Lists [1/2]

An **array** stores a sequence of data items, one after the other, in a contiguous block of memory. Example: `std::vector`.



A **Linked List** stores the same kind of data, but 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.



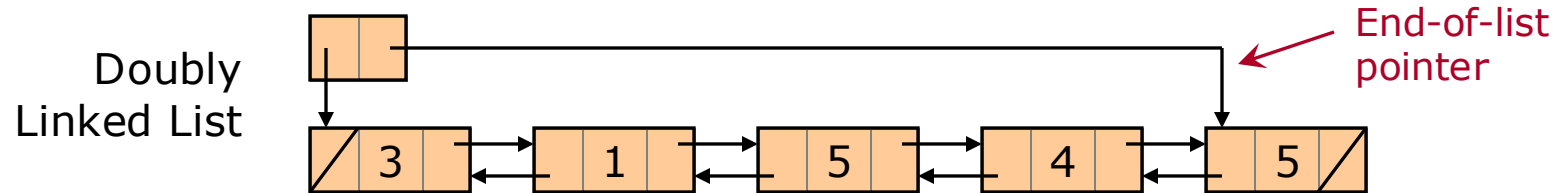
These pointers are the only way to find the next item. A Linked List is a forward-only sequential-access structure.

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

Review

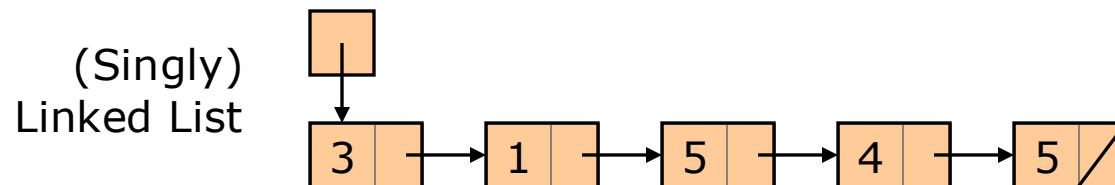
Arrays & Linked Lists [2/2]

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



A Doubly Linked List is a bidirectional sequential-access structure.

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



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) Base case
{
```

```
    if (a <= 1)
        return a == 1 ? b : 0;
```

Direct recursion

Recursive case

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

Indirect recursion

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

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 `fibonacci` that computes Fibonacci numbers.

See `fibonacci.cpp`.

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

Search Algorithms I

Search Algorithms I

Binary Search — Description [1/3]

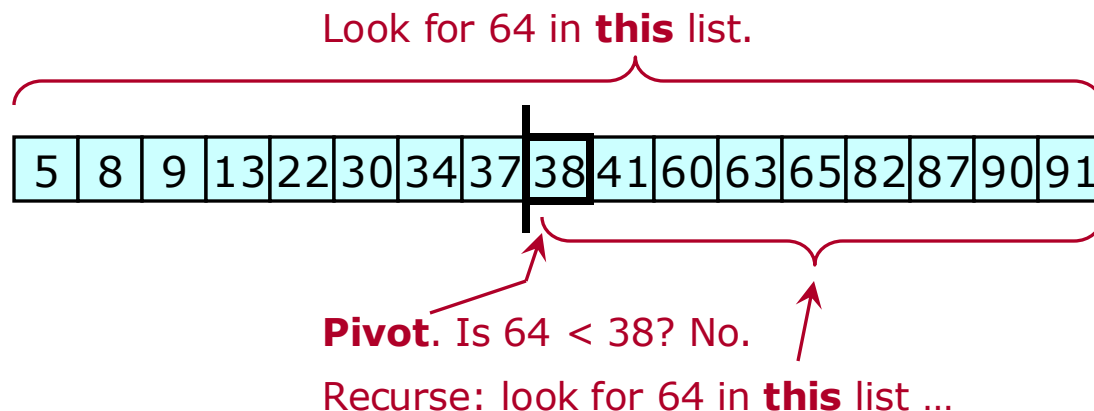
The **Binary Search** algorithm finds a given **key** in a **sorted list**.

- 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.

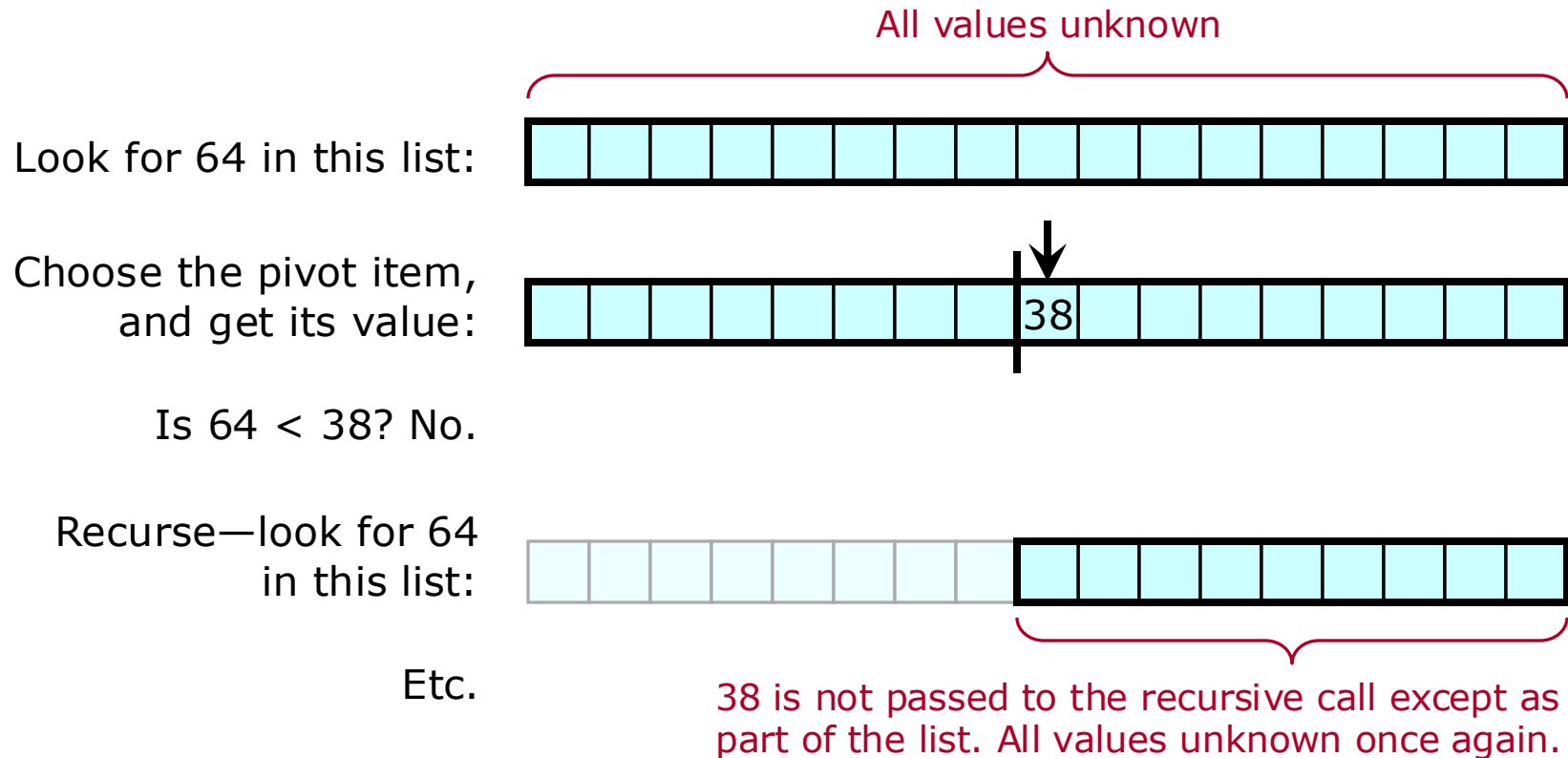


Search Algorithms I

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:



Search Algorithms I

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.

This idea will apply to many algorithms and data structures we cover.

Consider a list of customers of a business. Each is identified by a unique ID string, which is used as the key. The item for a customer also includes name, address, and order history.

Keys →

ID: bliu023	ID: fuda002	ID: smio344	...
Name: Uma Blint	Name: Alice Fudd	Name: Ole Smith	...
Addr: ...	Addr: ...	Addr:
OrderHist: ...	OrderHist: ...	OrderHist:

Since the keys are sorted, we can do Binary Search to find a particular key, and then retrieve the associated data.

Search Algorithms I

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.

Search Algorithms I

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.)

Search Algorithms I

Binary Search — CODE

TO DO

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

Done. See `binsearch1.cpp`.

Search Algorithms I

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 may make the base case more complicated. But when we do a search, the base case will only happen once; there may be many recursive calls.

Random-access iterators can do pointer-style arithmetic:

- `iter1 += 3;`
- `iter2 = iter1 + 3;`
- `n = iter1 - iter2;`

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

But we can get the same *results* for more general **forward iterators** with the following (all in `<iterator>`).

- `std::advance(iter, n)` is like `iter += n`.
- `std::next(iter, n)` is like `iter + n`.
- `std::distance(iter1, iter2)` is like `iter2 - iter1`.

These functions are fast for random-access iterators; they may be slower for other iterators.

Improvement: Replace pointer arithmetic the above functions.

↑ Allow the parameters to be forward iterators.

Is this really an improvement? Maybe. Regardless, `std::advance/next/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.
 - Replace pointer arithmetic with `std::advance/next/distance`. Allow the parameters to be forward iterators.

Done. `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!