

Parallel And Sequential Data Structures and Algorithms

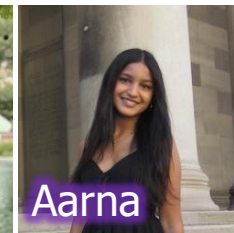
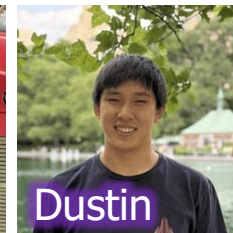
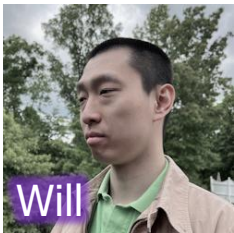
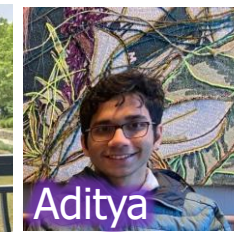
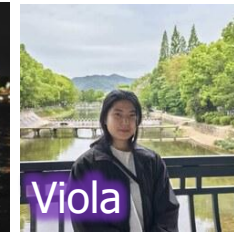
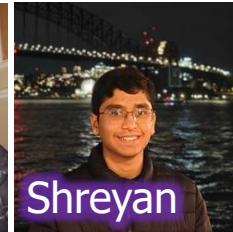
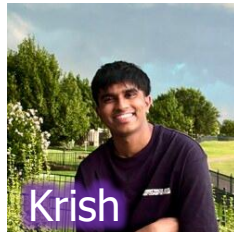
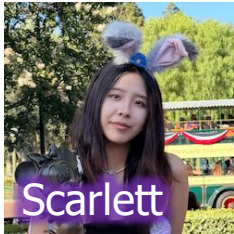
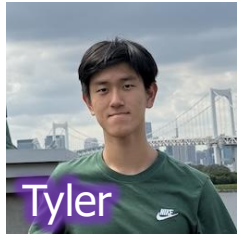
Introduction to 15-210

Learning Objectives

- Understand **what kind of course this is** (and how it differs from previous offerings)
- Understand **how we will write algorithms** (imperative *and* functional)
- Understand **how we reason about algorithms** (abstraction, cost, interfaces)

Course Logistics and Goals

Staff



Course Website and Platforms

- See **the website** for lecture notes, lecture slides, course policies, and resources: <https://www.cs.cmu.edu/~15210-f26/>
- We will use **Ed** for questions and announcements: <https://edstem.org/us/courses/102226/discussion>
- Homework will be submitted through **Gradescope**: <https://www.gradescope.com/courses/1348779>
- For TA office hours, use the Office Hours Queue (**OHQ**):
Link will be on the course website.

Classes

- **Lectures**

- Monday, Wednesday, 9:30am – 11:00am (Rashid)
- Some optional lectures on Friday 9:30am - 11:00am (Rashid)
- The Friday lecture slot is also used for **quizzes and midterms!**

- **Recitations**

- Tuesday, various times, see your schedule on SIO

Assessment

- **Lab Homeworks (Programming and Written)**
 - 11 labs, worth 25% in total
 - Released Tuesdays, due the following Monday
 - "Bucket" system: You only need 90% score to get full points!
 - No lab out on weeks that contain an exam
- **Quizzes**
 - Friday **18th Sep** (Week 4), **23rd Oct** (Week 8), **20th Nov** (Week 12)
 - Worth 15% in total (6% for your top two, 3% for your lowest)
- **Exams**
 - Midterms on Friday **2nd Oct** (Week 6) and **6th Nov** (Week 10)
 - Midterms are worth 15% each
 - Final exam worth 25%
- **Recitation Participation**
 - Worth 5%

Lab Assignments

You can choose to do your labs in either **SML** or **C++**

- Each lab has written problems or code problems or both
- You should typeset the written problems in LaTeX
 - A template is available on Ed
- Code problems are autograded on Gradescope
 - The autograder tests both your codes **correctness** and **performance**
 - We will manually grade your code to check that it meets the cost bound requirements of the problem

C++ in 15-210

- All labs can be completed in either **Parallel SML** or **C++**
- **Same ideas, different syntax:** C++ can be used in a *functional, parallel style*, mirroring the SML version.

Note: You do not need any advanced or systems-level C++.

- No memory management
- No pointers
- No OOP concepts (inheritance/overriding/virtual functions)
- No concurrency/atomics/threading
- No move semantics/perfect forwarding/resource management

Recitation Participation

- Starting from Recitation 2, we will assess your participation in recitation with a brand new system this semester
- Each Monday, we will give you a small number of quick concept-check questions
- You must **handwrite** answers to these questions and turn them in to your TA at the **start** of recitation
- Starting from Recitation 2, laptops will not be allowed in recitation to reduce distractions and give you practice working on paper. (Recitation 1 is a coding lab, so laptops are allowed.)

Imperative vs Functional



Functions

- In functional programming, functions usually correspond to mathematical functions: a map from inputs to outputs

```
fun f(n : int) -> int:  
  if (n <= 1): return 1  
  else: return f(n - 1) + f(n - 2)
```

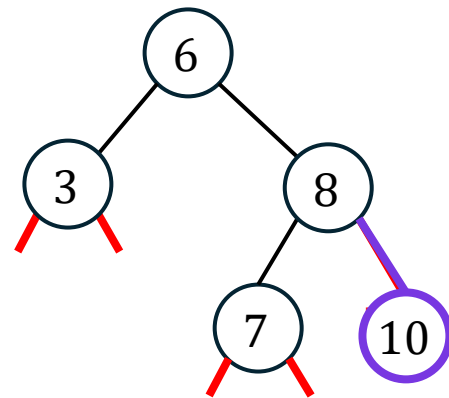
- Imperative functions can mutate or depend on shared (external) mutable state

```
x : int = 0  
y : int = 0  
  
fun f(n : int) -> int:  
  x ← 23  
  if (n <= 1): return 1;  
  else: return f(n - 1) + y * 3;
```

Definition (Pure function): A pure function is a function that always returns the same output given the same input and has no side effects.

Imperative Data Structures

```
// source: 15-122
tree* bst_insert(tree* T, entry e) {
    if (T == NULL) return leaf(e);
    int cmp = key_compare(entry_key(e), entry_key(T->data));
    if (cmp == 0) T->data = e;
    else if (cmp < 0) T->left = bst_insert(T->left, e);
    else T->right = bst_insert(T->right, e);
    return T;
}
```



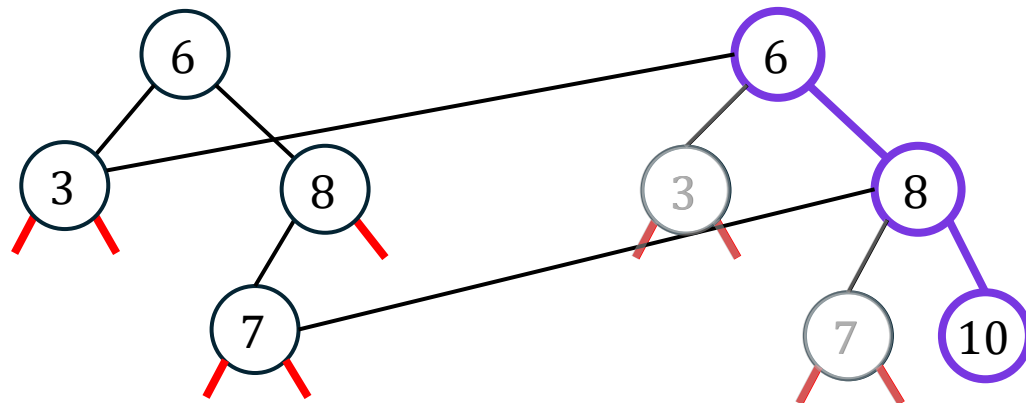
- Insertion into an imperative-style (mutable) data structure **modifies** the existing data structure

Example: `bst_insert(T, 10)`

- The right child of the 8-node is ***mutated*** to point to a new node containing 10

Functional Data Structures

```
(* source: 15-150 *)  
fun insert (k, v) Empty =  
  Node (Empty, (k, v), Empty)  
| insert (k, v) (Node (L, (k', v'), R)) =  
  case Key.compare (k, k') of  
    EQUAL   => Node (L, (k, v), R)  
  | LESS    => Node (insert (k, v) L, (k', v'), R)  
  | GREATER => Node (L, (k', v'), insert (k, v) R)
```



- Insertion into a functional data structure returns a **new data structure**
- The existing data structure remains unmodified

Example: `insert(10, _)`

- New nodes are created for each node along the path
- Old nodes not on the insertion path are **reused**

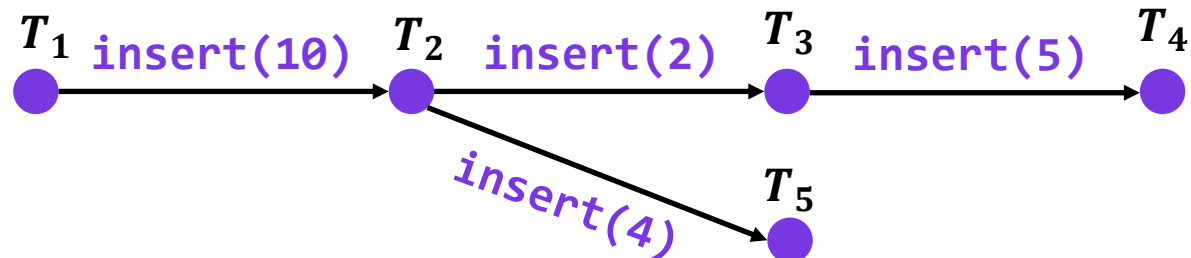
Persistence

- Functional data structures provide a useful property for free:

Definition (Persistent data structure):

A persistent data structure is a data structure that preserves the old versions of itself when it is updated.

Persistence is like having version-control history of your data structure!



Making Data Structures Persistent

James R. Driscoll

Comp. Sci. Dept., Carnegie-Mellon Univ.
Pittsburgh, PA 15218

Neil Sarnak

Courant Inst. of Mathematical Sciences
New York University
New York, New York 10012

Daniel D. Sleator

Comp. Sci. Dept., Carnegie-Mellon Univ.
Pittsburgh, PA 15218

Robert E. Tarjan

Comp. Science Dept., Princeton Univ.
Princeton, New Jersey 08544
and
AT&T Bell Laboratories
Murray Hill, NJ 07974

Abstraction: Interfaces and Abstract Data Types

Interfaces and ADTs

Definition (Interface): An interface specifies a **collection of operations** together with their intended behavior. It does not describe how those operations are implemented.

Definition (ADT): An **abstract data type** (ADT) is an interface that describes a data structure. It specifies the operations that can be performed, while leaving the underlying representation unspecified.

- An **implementation** provides algorithms for the operations in the interface
- A **data structure** is an implementation of an abstract data type, consisting of both a representation together with algorithms for the operations.

Why Interfaces?

- **Modularity:** Can improve or replace a data structure with a different one that implements the same interface and the code that uses it still works without needing changes
- **Reasoning:** Correctness proofs and runtime analysis need only reason about the interface and its guarantees, independent of the low-level implementation details (assuming the data structure is correct)
- **Reuse:** The same generic algorithm can be applied to multiple different types if they support the correct interface

The Sequence ADT

Definition (Sequence): A sequence of length n over elements of type T is an ordered collection of values that can be viewed as a mapping from the indices

$$\{0, 1, \dots, n - 1\} \rightarrow T$$

Interface (Sequence): A `sequence<T>` (with value type T) supports

- `nth(S : sequence<T>, i : int) -> T`:
returns the i^{th} element of the sequence S , abbreviated $S[i]$
- `length(S : sequence<T>) -> int`:
return the length of the sequence S , abbreviated $|S|$
- `subseq(S : sequence<T>, i : int, k : int) -> sequence<T>`:
returns a view of the subsequence of S starting at index i with length k

Array Sequences

- Assume that the sequences we construct are **ArraySequence<T>**, a contiguous fixed-size array, which supports $O(1)$ time operations
- This is the type we will assume is returned by the **tabulate** primitive, which constructs a sequence from a function

tabulate : (f : (int -> T), n : int) -> ArraySequence<T>

- **tabulate**(f, n) returns a sequence of length n where $S[i] = f(i)$, i.e.,

$[f(0), f(1), \dots, f(n-1)]$.

- We may also use Python-like syntax in our pseudocode, e.g., we may write

parallel [f(i) **for** i **in** 0...n-1]

Other Example ADTs

Interface (Dictionary): A `dictionary`<K,V> (key type K, value type V) has:

- **insert**(D : `dictionary`<K,V>, k : K, v : V) -> `void`:
add the given key, value pair $k:v$
- **find**(D : `dictionary`<K,V>, k : K) -> `option`<(K,V)>:
return the item with the given key k (NONE if it doesn't exist)
- **delete**(D : `dictionary`<K,V>, k : K) -> `void`:
delete the item with the given key k

- The most common implementations of dictionaries are:
 - **Hash tables** (common in imperative code)
 - **Binary search trees** (common in functional and imperative code)

Abstraction: Models of Computation and Cost Models

Analyzing Costs of Algorithms

- We make claims like "Insertion sort runs in $O(n^2)$ time" and "Merge Sort runs in $O(n \log n)$ time"

Question: How are we measuring "time"?

- In 15-122, you settled on *"number of execution steps"*
- "Execution steps" include integer arithmetic, conditionals, function calls, reading/writing memory, etc.
- One important caveat: we should not allow *arbitrary-precision arithmetic* in a *single step*

Word RAM Model

Definition (Word RAM model):

- Memory values are **w -bits long**,
 - Operations on w -bit values, e.g., arithmetic, comparison, branching, reading/writing any memory location, cost $O(1)$,
 - w is at least $\log n$ for problem inputs of size n .
-
- Same in spirit as the "*number of execution steps*" from 15-122
 - Only difference is the restriction on the precision to some **fixed number of bits w**

Why Limit Precision?

<pre>int product = 1; for (int i = 1; i <= n; i++) product *= i;</pre>	<pre>product = 1 for i in range(1,n+1): product *= i</pre>
---	--

Question: What goes wrong with this C code and Python code?

- In C, the product overflows a 64-bit integer for just $n \geq 21$
- In Python, it takes **more than constant space**
- In summary: don't write algorithms that rely on higher precision arithmetic than the precision of the input

Parallel Cost Models

- The Word RAM model is purely sequential. To analyze parallel algorithms, we will need to extend the model

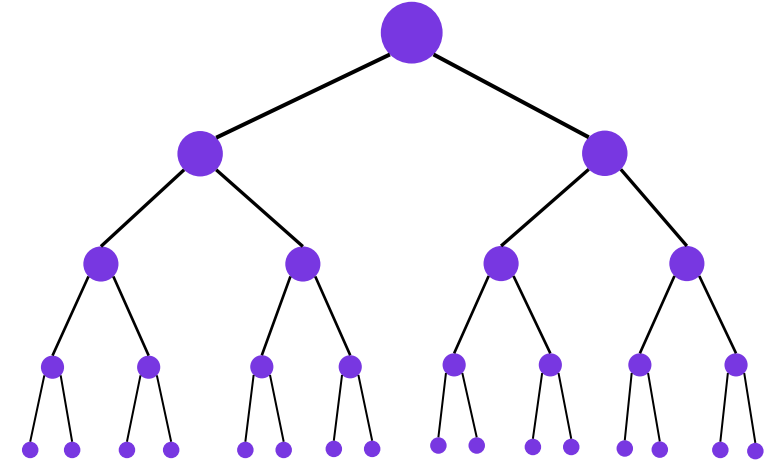
Definition (Fork-Join RAM): Extends the word RAM with a **fork** instruction.

- fork creates a fixed number of *child computations* that may execute in parallel.
 - The parent suspends at the fork point and resumes only after all its children have completed; the resumption is called the **join point**
- Child computations may perform forks, allowing parallelism to be nested.

Question: How much abstraction is this compared to real computer?

Nested Parallelism as an Abstraction

- Think of nested parallelism as having *infinitely many processors!*
- A process can always fork into more, and this can continue recursively (e.g., divide-and-conquer style). No limit.



Remark: Deciding which instructions run on which CPU core is called *scheduling*. Your OS also has a scheduler for running processes.

- Nested parallelism allows algorithms to express parallel structure without specifying *how* computations are assigned to processors

What Do We Measure?

- For sequential algorithms, we just measured one cost ("*time*")
- In 15-150, you learned that to describe the cost of a parallel algorithm, we need **two numbers**

Definition (Work): The **work** of a parallel algorithm is sum of the costs of all instructions it executes, across all parallel branches.

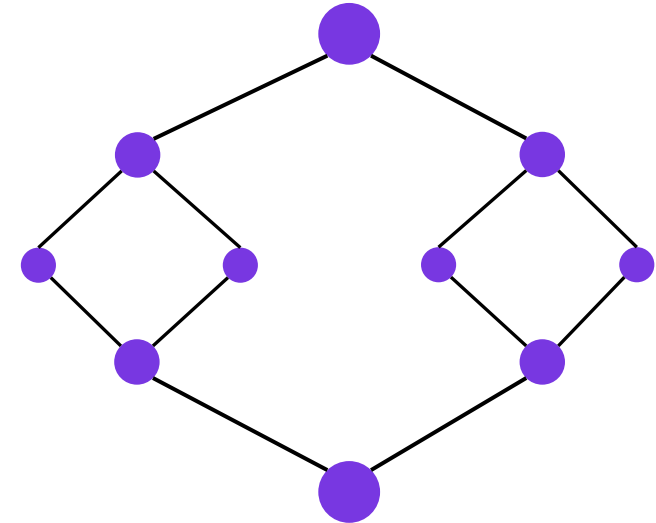
Intuition: Work is the cost of the algorithm on one processor (i.e. no parallelism)

Definition (Span): The **span** of a parallel algorithm is the cost of the longest chain of dependent computations

Intuition: Span is the cost of the algorithm with infinitely many processors!

Example: Parallel Sum (Reduce+)

```
fun sum(S : sequence<int>) -> int:
  match length(S) with:
  case 0: return 0           // Empty sequence
  case 1: return S[0]        // Singleton sequence
  case _:
    L, R = split_mid(S) // Helper function
    Lsum, Rsum = parallel (sum(L), sum(R))
    return Lsum + Rsum
```



Analysis via recurrence relation:

$W_{\text{sum}}(n) = 2W_{\text{sum}}(n/2) + O(1)$ which solves to $O(n)$
 $S_{\text{sum}}(n) = S_{\text{sum}}(n/2) + O(1)$ which solves to $O(\log n)$

Less formal analysis: The recursion has depth $O(\log n)$ and does constant work per node, so the span is $O(\log n)$.

Why "infinitely many processors"?

Question: most CPUs have neither one nor infinitely many processors... so why are these quantities useful?

- **Unsatisfying answer:** Span is a natural measurement for nested parallel algorithms since they are described as if there are infinite processors
- **Better answer:** It turns out that by measuring the work and span, we can derive the cost of the algorithm for any number of processors!

Theorem (Brent's Theorem): An algorithm with work W and span S can be scheduled on a P -processor machine in $O(\max(W/P, S))$ time

*Brent's Theorem essentially proves that nested parallelism is a **good abstraction!***

Summary

- You can do your homework in either **Parallel SML or C++!**
- Functional data structures differ from imperative ones by avoiding **mutation** and as a result are naturally **persistent** which can have useful algorithmic applications
- Interfaces and ADTs are useful for specifying operations without committing to a concrete implementation
- Our chosen model for parallel algorithms is **nested fork-join parallelism**, analyzed by their **work** and **span**