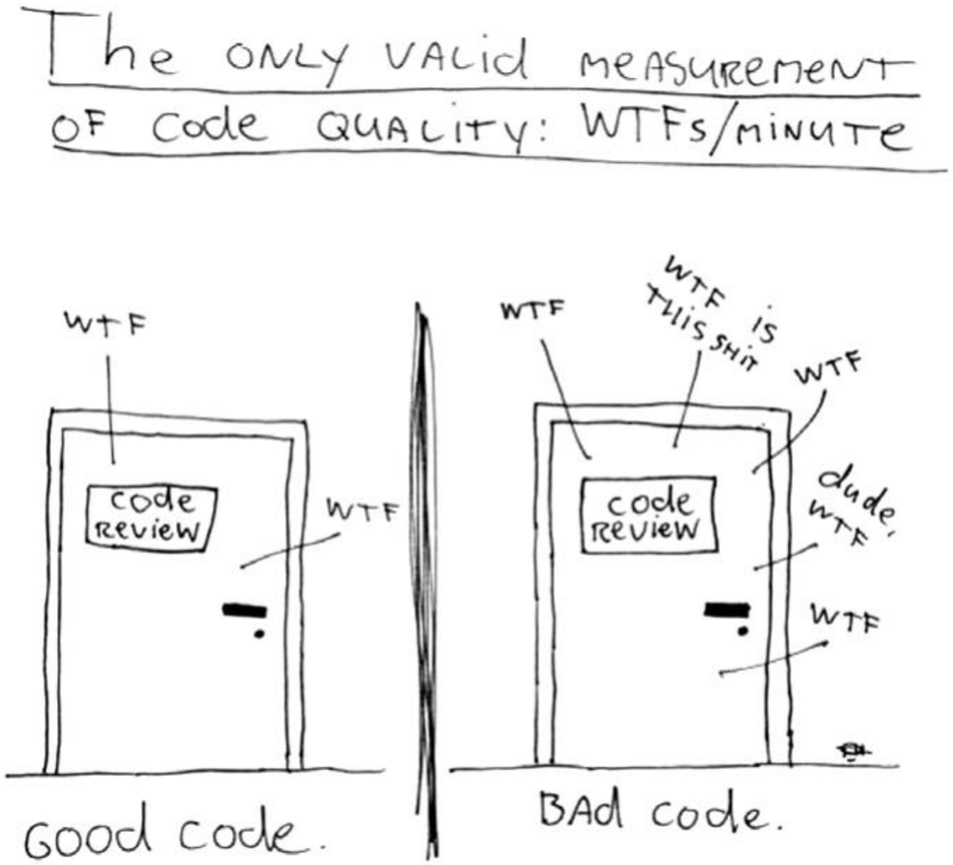
소프트웨어 공학

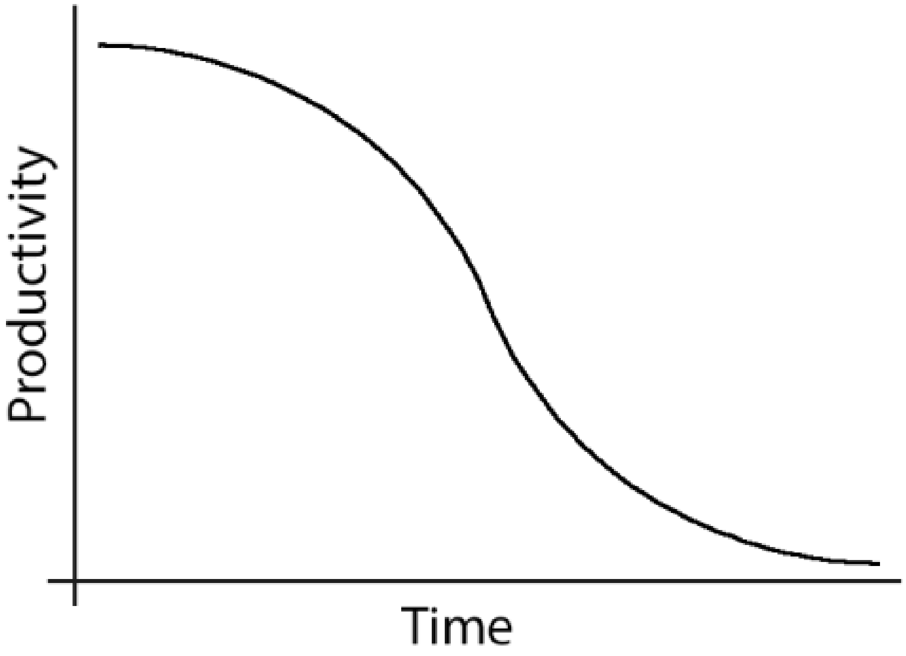
Dr. Young-Woo Kwon

좋은 코드란?



The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.

좋은 코드란?



이해하기 쉬운 코드

좋은 코드

좋은 코드 > 나쁜 코드 + 좋은 주석

주석을 추가하는 이유?

코드 품질이 나쁘기 때문에!

좋은 주석

- 법적인 주석
- 정보를 제공하는 주석
- 의도를 설명하는 주석
- 의미를 명료하게 밝히는 주석
- 결과를 경고하는 주석
- TODO 주석
- 중요성을 강조하는 주석

나쁜 주석

- 주절거리는 주석
- 같은 이야기를 중복하는 주석
- 오해할 여지가 있는 주석
- 의무적으로 다는 주석
- 이력을 기록하는 주석
- 있으나 마나 한 주석
- 위치를 표시하는 주석
- 닫는 괄호에 다는 주석
- 공로를 돌리거나 저자를 표시하는 주석
- 주석으로 처리한 코드
- HTML 주석
- 너무 많은 정보

주석에 담아야 하는 대상

우리는 코드를 작성할 때 머릿속에 귀중한 정보가 있다. 그런데 다른 사람이 그 코드를 보면 그런 귀중한 정보는 없다. 그들이 가진 정보라곤 눈앞에 있는 코드뿐이기에

코드를 읽는 사람이 코드를 작성한 사람만큼 코드를 잘 이해할 수 있게 도울 수 있어야 한다.

주석에 담아야 하는 대상

설명하지 말아야 할 것

```
// 클래스 Account를 위한 정의
class Account {
    public:
        // 생성자
        Account();

        // profit에 새로운 값을 설정
        void SetProfit(double profit);

        // 이 어카운트의 profit을 반환
        double GetProfit();
}
```

코드에서 유추할 수 있는 내용의 주석은 피할 것!

주석에 담아야 하는 대상

설명하지 말아야 할 것

```
# 두 번째 '*' 뒤에 오는 내용을 모두 제거한다.
name = '*'.join(line.split('*')[:2]);
```

코드에서 유추할 수 있는 내용의 주석은 피할 것!

주석에 담아야 하는 대상

설명 자체를 위한 설명을 달지 말 것

```
// 주어진 이름과 깊이를 이용해서 서브트리[h1]에 있는 노드를 찾는다.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

```
// 주어진 'name'으로 노드를 찾거나 아니면 NULL을 반환한다.
// 만약 depth <= 0이면 'subtree'만 검색된다.
// 만약 depth == N이면 N레벨과 그 아래만 검색된다.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

더 중요한 세부사항을 적는 것이 낫다

주석에 담아야 하는 대상

자신의 생각을 기록하는 것 (정보 제공)

```
// 놀랍게도, 이 데이터에서 이진트리는 해시테이블보다 40% 정도 빠르다.
// 해시를 계산하는 비용이 좌/우 비교를 능가한다.
```

Too Much Information

```
/*
  RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
  Part One: Format of Internet Message Bodies section 6.8.
  Base64 Content-Transfer-Encoding
  The encoding process represents 24-bit groups of input bits
  as output strings of 4 encoded characters. Proceeding from
  left to right, a 24-bit input group is formed by
  concatenating 3 8-bit input groups.
  These 24 bits are then treated as 4 concatenated 6-bit
  groups, each of which is translated into a single digit in
  the base64 alphabet.
  When encoding a bit stream via the base64 encoding, the bit
  stream must be presumed to be ordered with the most-
  significant-bit first.
 */
```

주석에 담아야 하는 대상

코드에 있는 결함을 설명하라

```
// TODO: 더 빠른 알고리즘을 사용하라.
```

표시	보통의 의미
TODO:	아직 하지 않는 일
FIXME:	오동작을 일으킨다고 알려진 코드
HACK:	아름답지 않은 해결책
XXX:	위험! 여긴 큰 문제가 있다.

주석에 담아야 하는 대상

- 상수에 대한 설명

```
NUM_THREADS = 8
```

```
NUM_THREADS = 8 # 이 상수값이 2 * num_processors보다 크거나 같으면 된다.
```

읽기 쉬운 흐름제어 만들기

조건문에서의 인수의 순서

```
if (length >= 10)
```

vs

```
if (10 <= length)
```

왼쪽값은 유동적인 값
오른쪽은 고정적인 값

읽기 쉬운 흐름제어 만들기

if/else 블록의 순서 (긍정 먼저)

```
if (a == b) {
    // 첫번째 경우
} else {
    // 두번째 경우
}
```

vs

```
if (a != b) {
    // 두번째 경우
} else {
    // 첫번째 경우
}
```

관심이 있는 것 먼저

```
if (!url.HasQueryParameter("expand_all")) {
    response.Render(items);
    ...
} else {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
    }
    ...
}
```

vs

```
if (url.HasQueryParameter("expand_all")) {
    for (int i = 0; i < items.size(); i++) {
        items[i].Expand();
    }
    ...
} else {
    response.Render(items);
    ...
}
```

읽기 쉬운 흐름제어 만들기

함수 중간에서 반환하기

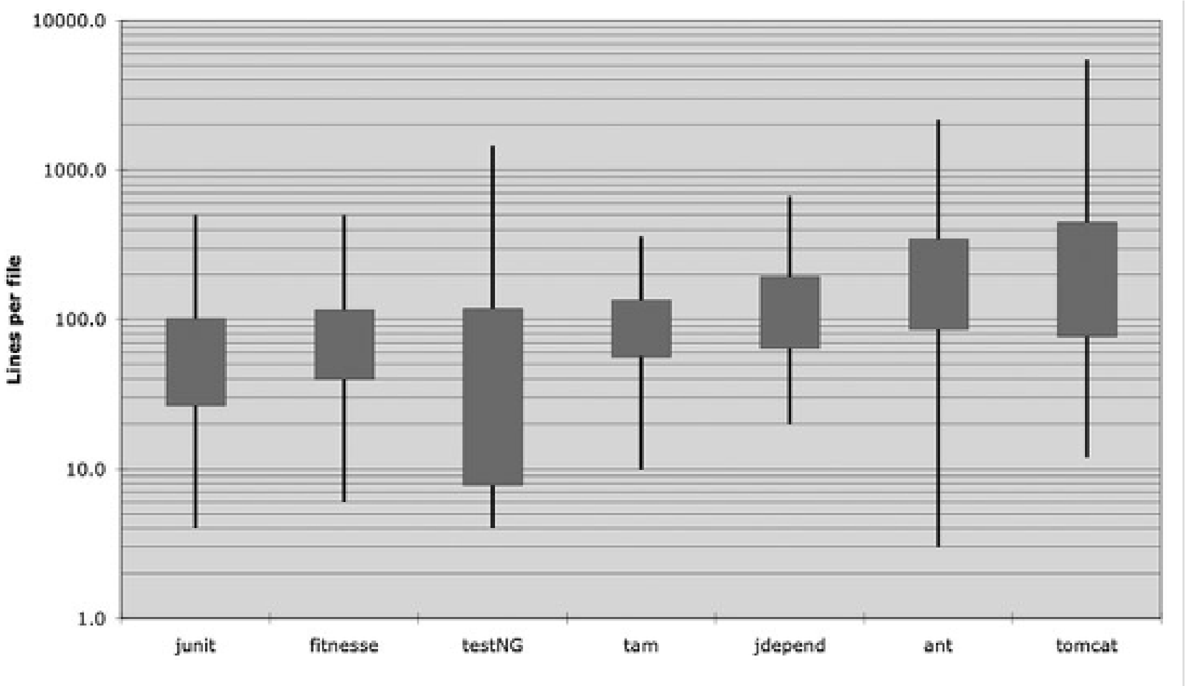
```
public boolean Contains(String str, String substr) {
    if (str == null || substr == null) return false;
    if (substr.equals("")) return true;
    // ...
}
```

클린업 코드 실행을 하려면?

언어	클린업 코드를 위한 관용적 구조
C++	destructors
자바, 파이썬	try finnally
파이썬	with
C#	using

형식 맞추기

- 소스코드의 길이
  - 100줄 내외의 짧은 길이

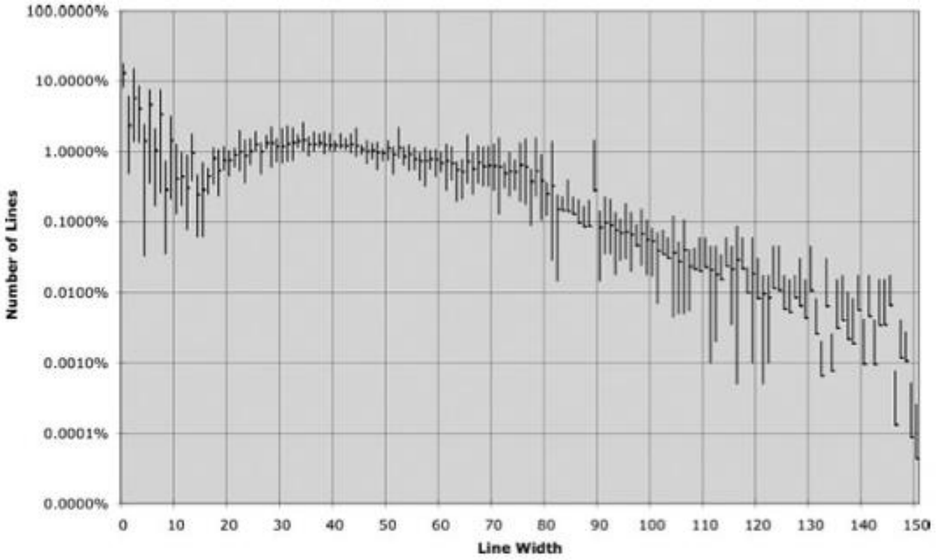


형식 맞추기

- 이야기하는 코드
- 개념은 빈 행으로
  - 패키지, import, 각 함수
- 세로 밀집도
  - 밀접한 코드 행은 세로로 가까이
- 변수 선언
  - 사용하는 위치에 최대한 가까이

형식 맞추기

- 가로 형식 맞추기
  - 80 → 100 → 120



예외 처리

- 가독성을 높이는 예외처리

```java
public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1); // Check the state of the device
    if (handle != DeviceHandle.INVALID) {
        // Save the device status to the record field
        // retrieveDeviceRecord(handle);
        // If not suspended, shut down
        if (record.getStatus() != DEVICE_SUSPENDED) {
            pauseDevice(handle);
            clearDeviceWorkQueue(handle);
            closeDevice(handle);
        } else {
            logger.log("Device suspended. Unable to shut down");
        }
    } else {
        logger.log("Invalid handle for: " + DEV1.toString());
    }
}
```

```java
public void sendShutDown() {
    try {
        tryToShutDown();
    } catch (DeviceShutDownError e) {
        logger.log(e);
    }
}
```

오류 코드를 사용하지 말 것

Null을 반환하거나 전달하지 말 것

예외처리

- 예외는 진짜 예외 상황에서만 사용
- 복구할 수 있는 상황에는 검사 예외를, 프로그래밍 오류에는 런타임 예외를 사용
- 필요 없는 검사 예외 사용은 피할 것
- 예외를 무시하지 말 것
  - Catch 블록을 비워두는 경우

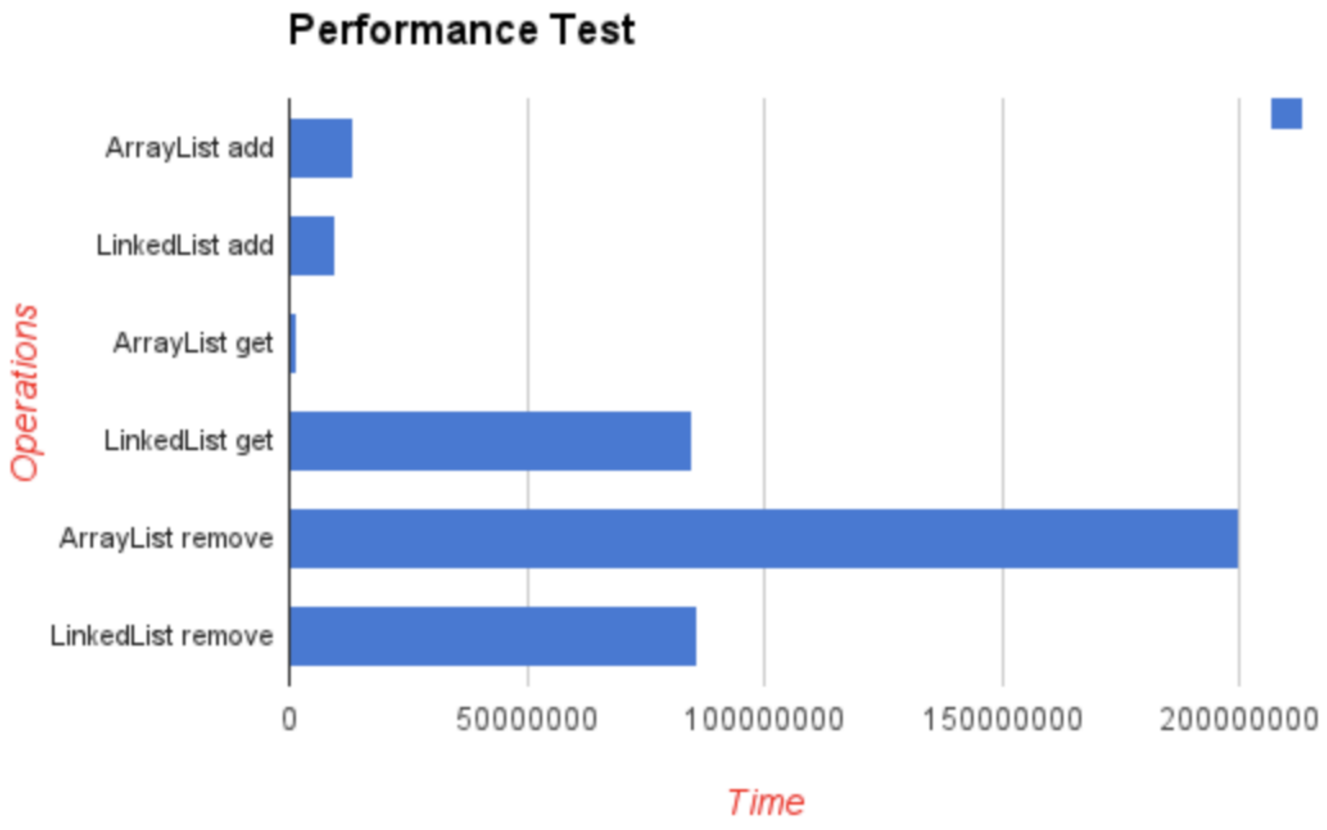
자료구조

자료 구조의 선택 기준

- 시간 복잡도
- 공간 복잡도
- 동시성

시간 복잡도

- LinkedList vs. ArrayList

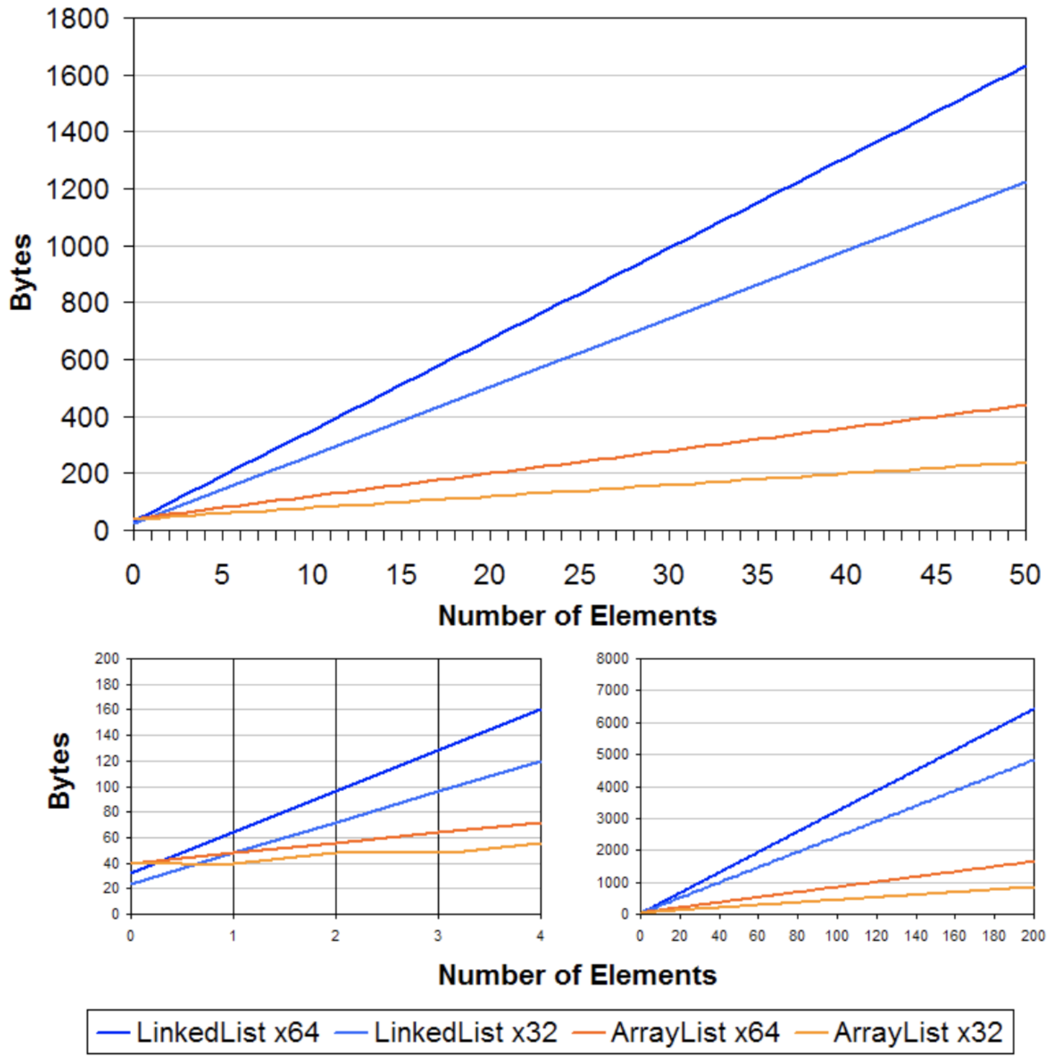


시간 복잡도와 N의 크기 관계

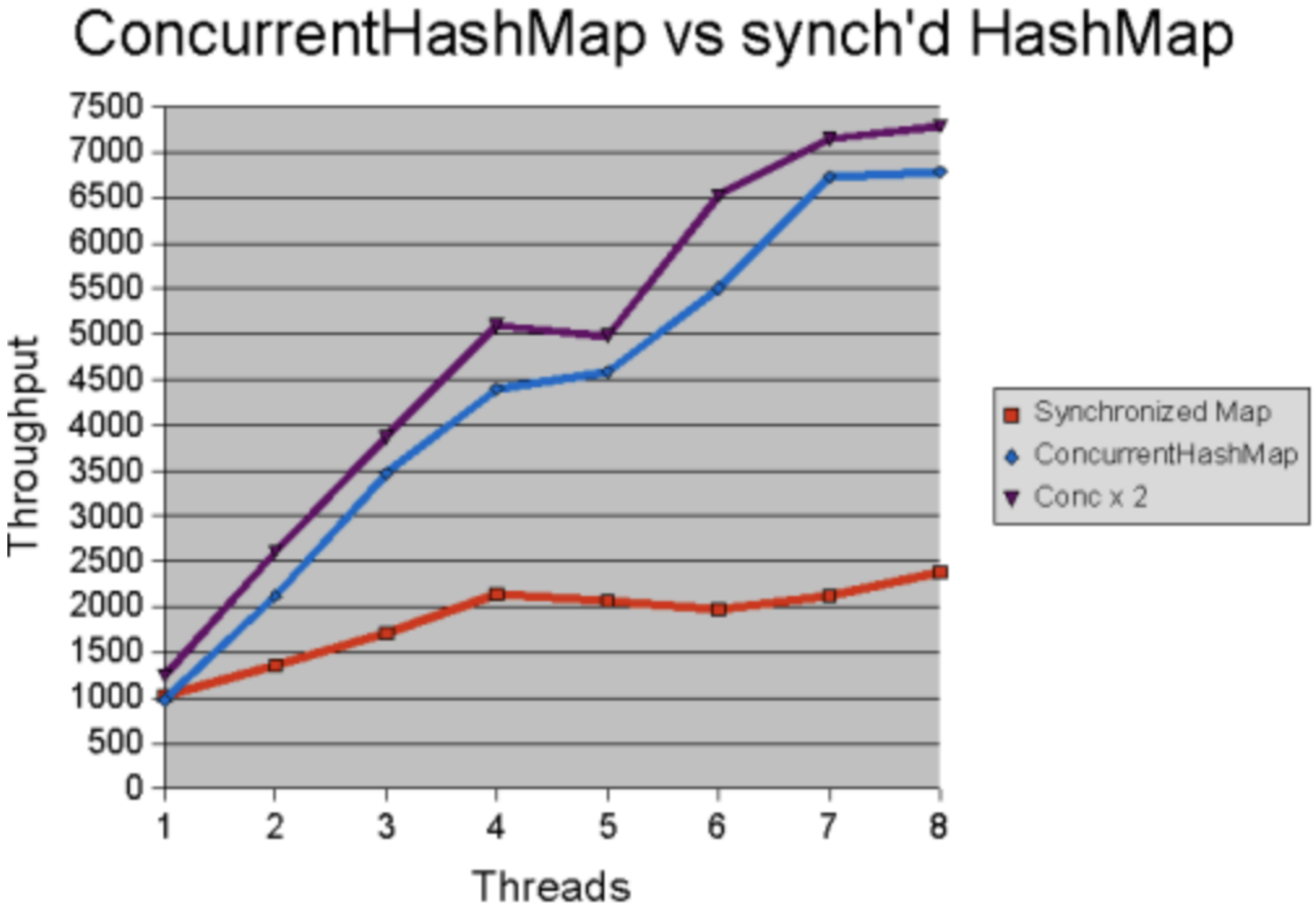
N의 크기	허용 시간복잡도
$N \leq 11$	$O(N!)$
$N \leq 20$	$O(2^N)$
$N \leq 100$	$O(N^4)$
$N \leq 500$	$O(N^3)$
$N \leq 3,000$	$O(N^2 lgN)$
$N \leq 5,000$	$O(N^2)$
$N \leq 1,000,000$	$O(NlgN)$
$N \leq 10,000,000$	$O(N)$
그 이상	$O(lgN), O(1)$

공간 복잡도

- LinkedList vs. ArrayList



동시성 (Concurrency)



자료구조 논쟁

- 코딩 테스트와 면접
  - 신입 면접 vs. 경력 면접
- 자료구조/알고리즘 vs. 디자인 패턴
  - 닭이 먼저? 알이 먼저?
  - 둘 다 중요하지만 우선 순위를 매긴다면?

- https://okky.kr/article/340883 : 자료구조 꼭 필요한가요?
- https://okky.kr/article/396435 : 개인적으로 알고리즘 논란에 민감한 이유

자료구조의 종류

- 비트 마스크
- 선형 자료구조
  - 동적 배열, 연결 리스트
- 큐, 스택, 데크
- 해싱
- 트리
  - 이진 검색 트리, 구간 트리, 트라이
- 그래프
  - 깊이 우선 탐색, 너비 우선 탐색, 최단 경로 알고리즘, 최소 스패닝 트리

비트 마스크

- 이진수 표현을 자료 구조로 쓰는 기법
  - 더 빠른 수행 시간
  - 더 간결한 코드
  - 더 작은 메모리 사용량
- 비트 마스크의 연산
  - &, |, ^, ~, <<, >>
- 비트 마스크의 응용
  - 집합 표현 및 연산
    - 정수 1, 4, 5, 6, 7, 9를 가지는 집합

    →2^1 + 2^4 + 2^5 … = 754

C++과 Java에서의 비트 마스크

- C++
  - bitset
- Java
  - Java.util.BitSet
    - set, get, flip, bitwise

```
//Construct 03: Construct it from string
bitset<8> bitset3(string("11111100"));
cout << "Constructor with string parameter.Content of bi
    << bitset3 << endl;
```

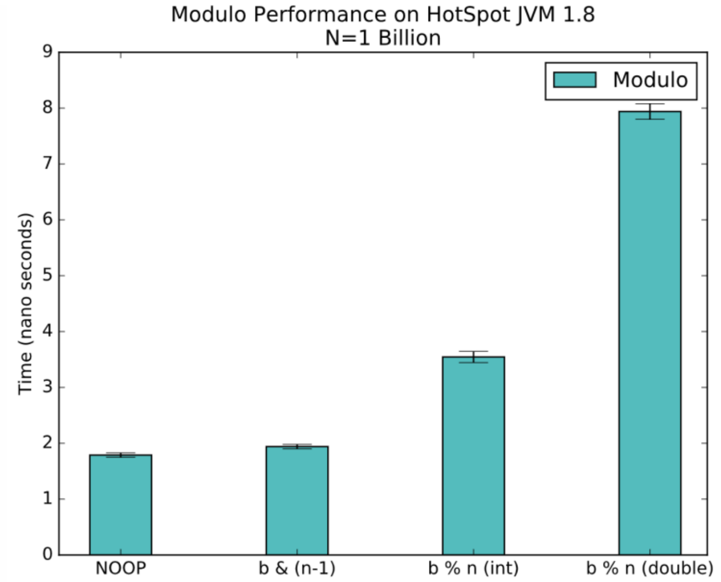
```
bitset2.set(4); cout << bitset2 << endl;
bitset2.set(7); cout << bitset2 << endl;
```

```
BitSet bitset = new BitSet(8);

// assign values to bitset1
bitset.set(0);
bitset.set(1);
bitset.flip(2,5);
```

실전 비트 마스크

- 컴파일러 최적화

Original Calculation	Replacement Calculation
y = x / 8	y = x >> 3
y = x * 64	y = x << 6
y = x * 2	y = x << 1
y = x * 15	y = (x << 4) - x



- 기타 테크닉
  - http://graphics.stanford.edu/~seander/bithacks.html

선형 자료 구조

- 동적 배열
  - 배열의 특징을 가짐
    - 메모리의 연속된 위치에 저장 → 캐시의 효율성과 직결
    - 주어진 위치에 대한 원소 반환, 변경 연산이 O(1)에 수행 가능
  - 추가적인 특징
    - 배열의 크기를 변경할 수 있음 → O(n)
    - 배열의 마지막에 추가할 경우 → O(1)
  - vector (C++), ArrayList (Java)
- 연결 리스트
  - 배열 원소들의 순서를 유지하면서 임의의 위치에 원소 삽입, 삭제를 O(1)에 수행
  - list (C++), LinkedList (Java)

동적 배열 vs. 연결 리스트

- 시간 복잡도

	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)

큐, 스택, 데크

- 큐 (FIFO), 스택 (LIFO), 데크
  - 연결 리스트를 통한 구현
    - 양쪽 끝에서의 삽입/삭제가 상수 시간에 가능
    - 하지만, 노드의 할당, 삭제, 접근 등에 시간이 걸림
  - 동적 배열을 통한 구현
    - 스택의 경우 쉽게 구현 가능
    - 그러나 큐/데크의 경우 삽입/삭제 시 시간이 O(n) 시간 소요
      - 해결 방안?

      →Circular Buffer
  - stack, queue (C++)
  - java.util.Stack/java.util.Queue (Java)

해싱

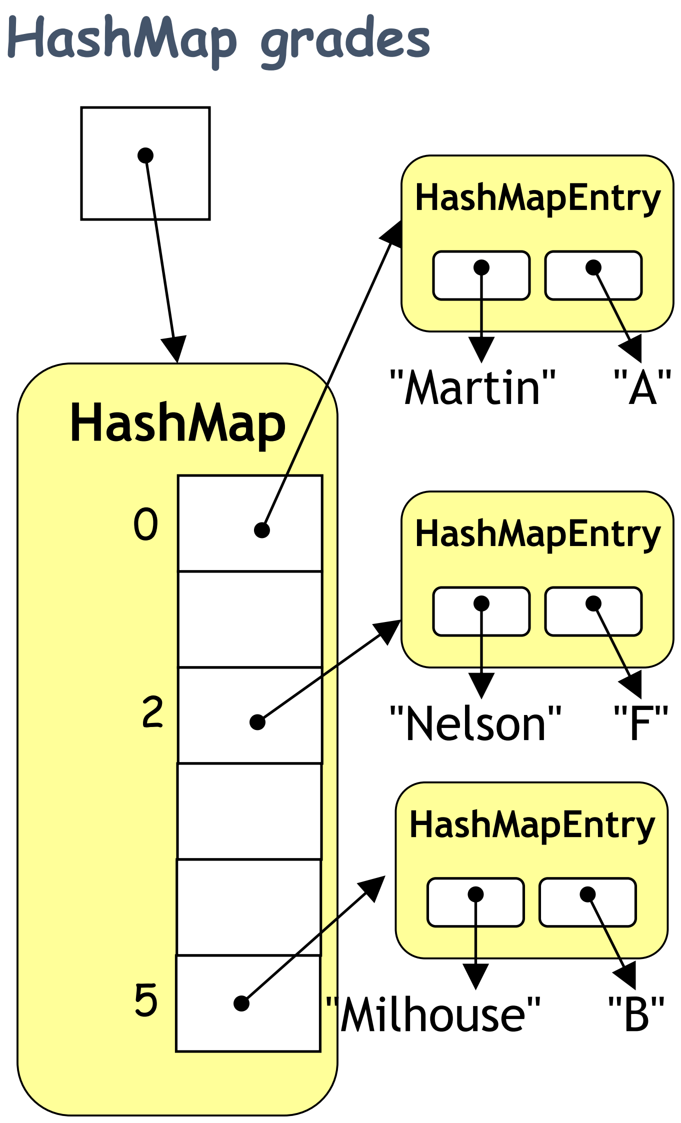
- HashMap, HashTable, TreeMap, LinkedHashMap, ConcurrentHashMap in Java
- map, set, hash_map, hash_set, unordered_map, unordered_set in C++

HashMap, HashTable 등 비교

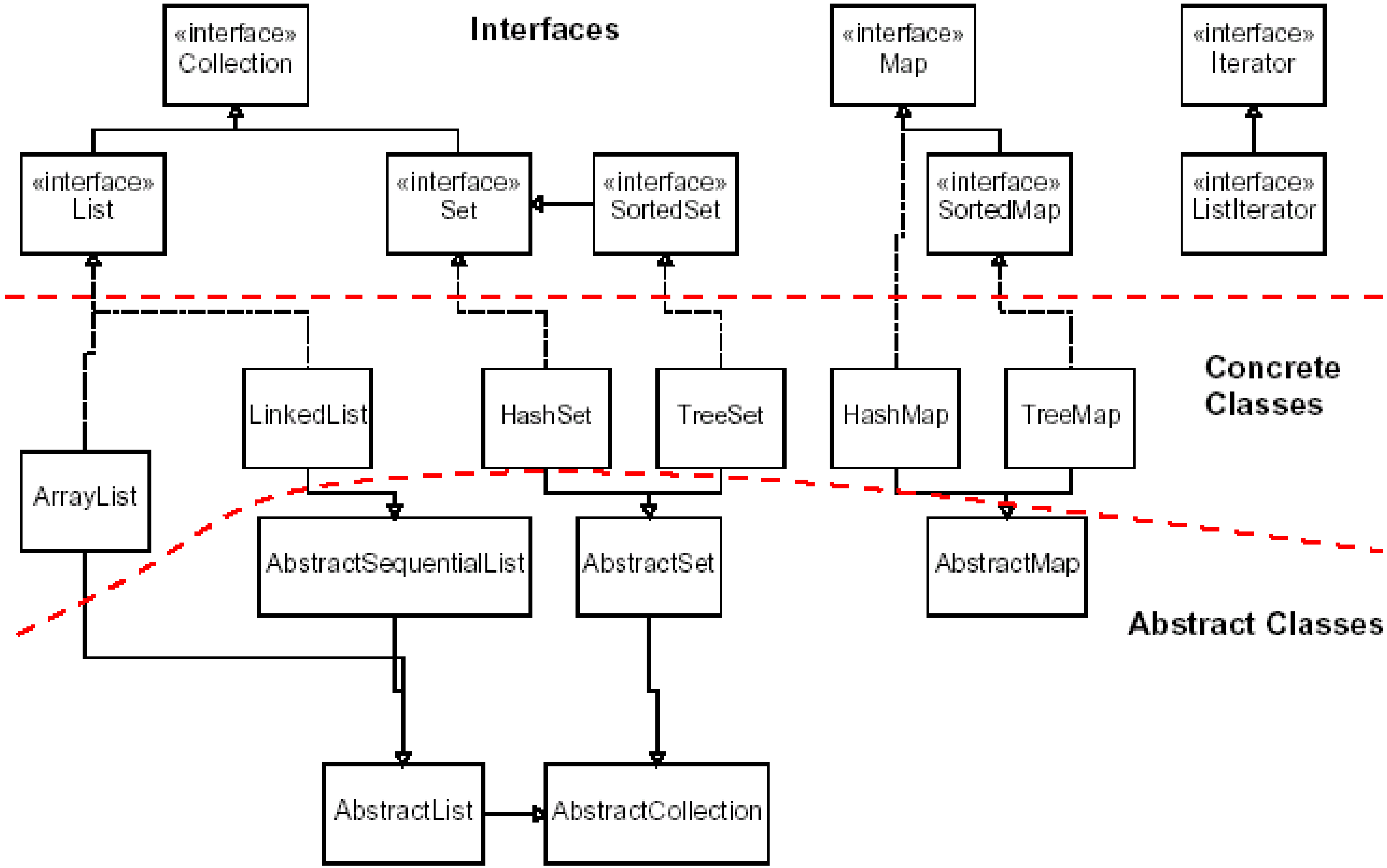
Property	HashMap	TreeMap	LinkedHashMap	HashTable
Iteration Order	Random	Sorted according to natural **order of keys**	Sorted according to the **insertion order**.	Random
Efficiency: Get, Put, Remove, ContainsKey	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$
Null keys/values	allowed	Not-allowed*	allowed	Not-allowed
Interfaces	Map	Map, SortedMap, NavigableMap	Map	Map
Synchronized	Not instead use Collection.synchronizedMap(new HashMap())			Yes but prefer to use ConcurrentHashMap
Implementation	Buckets	Red-Black tree	HashTable and LinkedList using doubly linked list of buckets	Buckets
Comments	Efficient	Extra cost of maintaining TreeMap	Advantage of TreeMap without extra cost.	Obsolete

HashMap example

```
HashMap grades = new HashMap();
grades.put("Martin", "A");
grades.put("Nelson", "F");
grades.put("Milhouse", "B");

// What grade did they get?
System.out.println(grades.get("Nelson"));
System.out.println(grades.get("Martin"));

grades.put("Nelson", "W");
grades.remove("Martin");

System.out.println(grades.get("Nelson"));
System.out.println(grades.get("Martin"));
```



Java Collections Framework



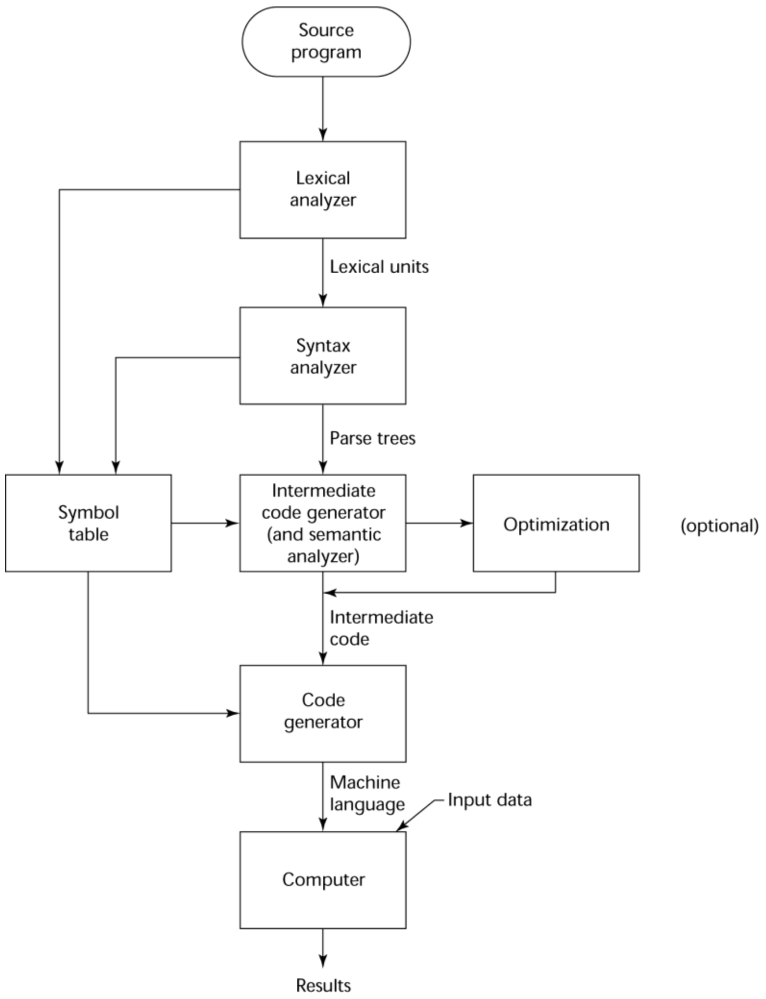
HashMap/HashTable 구현?

- 해싱
- 객체의 동일성
- 충돌

컴파일러

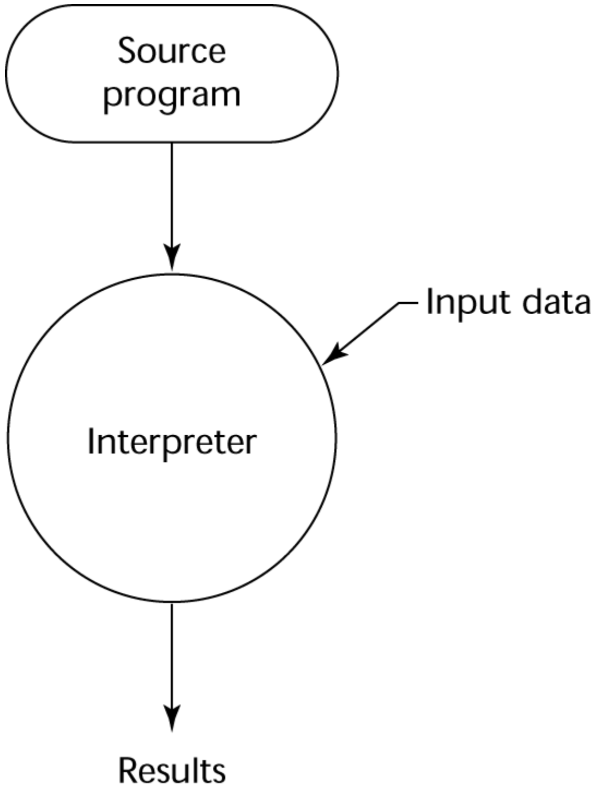
컴파일 과정

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:



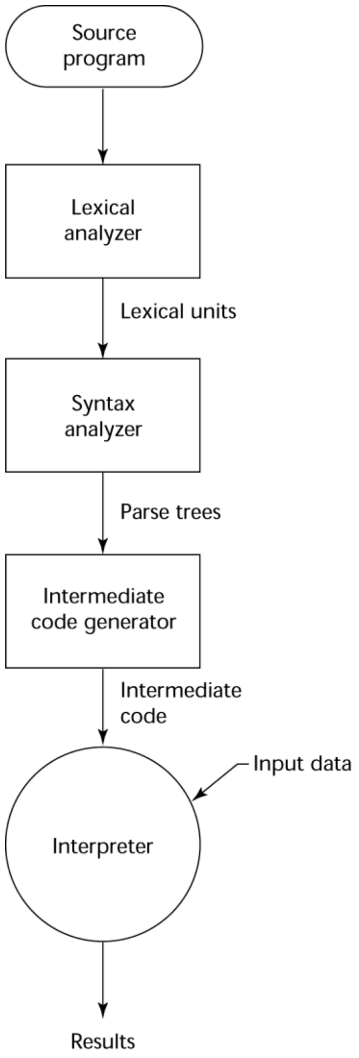
인터프리터

- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
  - Use: Small programs or when efficiency is not an issue
  - Examples?



인터프리터

- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters

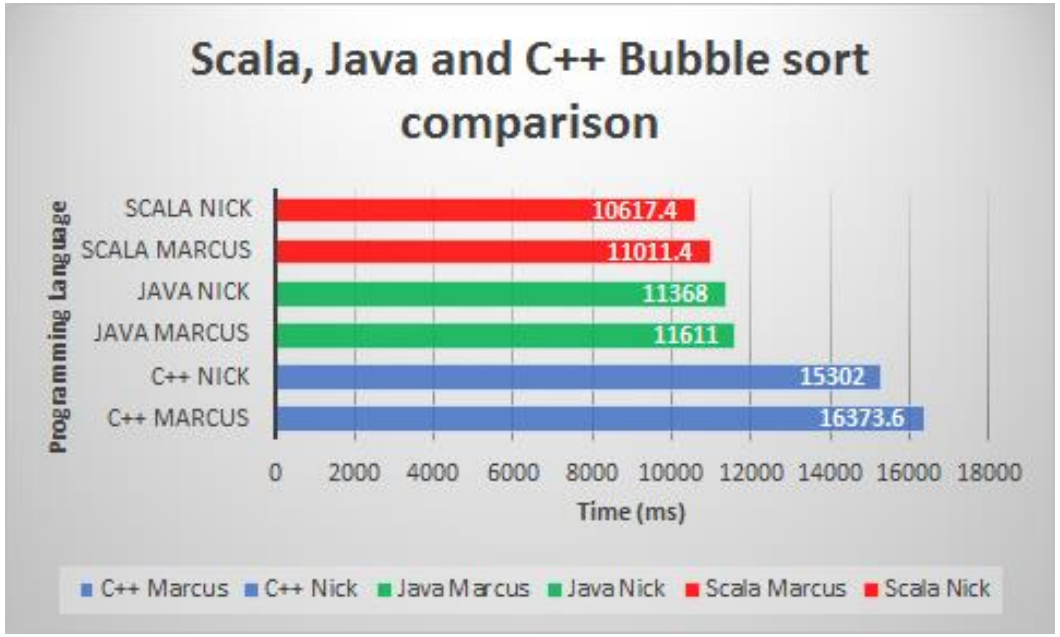


JIT (Just-in-time) compilation

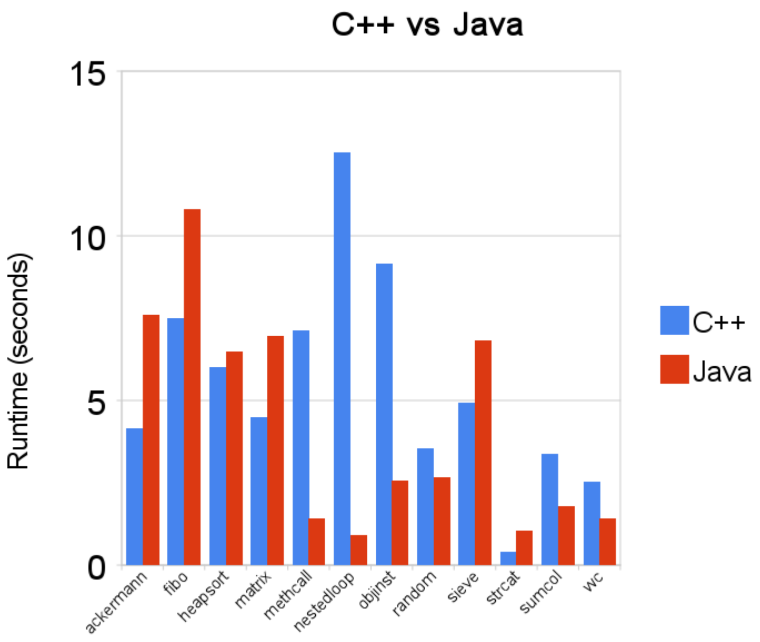
- JIT (Just-in-time) compilation
  - *“Compilation done during execution of a program (at run time) rather than prior to execution” - Wikipedia*
  - *“JIT code can in some cases offer better performance than static compilation, as many optimizations are only feasible at run-time” - Wikipedia*
- JIT compilation in JVM
  - A Java compiler compiles high level Java source code to Java bytecode readable by JVM
  - JVM compiles bytecode at runtime into machine readable instructions as opposed to interpretting
  - run compiled machine readable code

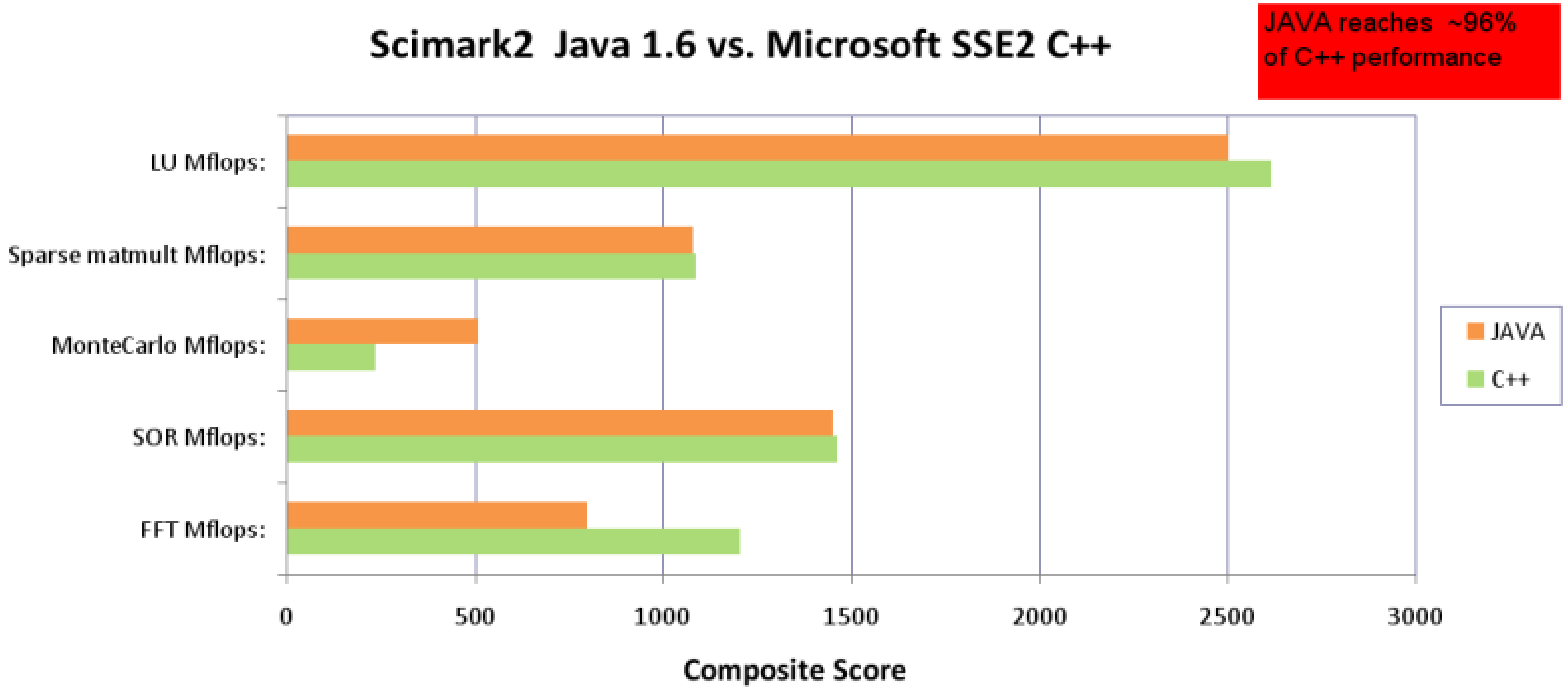
your naively written Java code outperform
your naively written C++ code

Your naively written Java code outperform your naively written C++ code



벤치마크


C++ vs Java
15
10
5
0
Runtime (seconds)
ackermann
fibo
heapsort
matrix
methcall
nestedloop
objinst
random
sieve
strcat
sumcol
wc
C++
Java


Scimark2 Java 1.6 vs. Microsoft SSE2 C++
JAVA reaches ~96% of C++ performance
LU Mflops:
Sparse matmult Mflops:
MonteCarlo Mflops:
SOR Mflops:
FFT Mflops:
0
500
1000
1500
2000
2500
3000
Composite Score
JAVA
C++

Inlining

- Inlining is the process by which the trees of smaller methods are merged, or "inlined", into the trees of their callers. This speeds up frequently executed method calls.

Local Optimizations

- Local optimizations analyze and improve a small section of the code at a time. Many local optimizations implement tried and tested techniques used in classic static compilers.
    - Algebraic simplification
        - x := x * 0 ⇒ x := 0
        - x := x * 8 ⇒ x := x << 3
    - Constant folding
        - x := 2 + 2 ⇒ x := 4
    - Eliminating unreachable code
    - Common subexpression elimination
        - a = b * c + g; d = b * c * e;

        → tmp = b* c; a = tmp + g; d = tmp * e;
    - Copy propagation
        - y = x; z = 3 + y → z = 3 + x

Null Check Elimination

```java
private static void runSomeAlgorithm(Graph graph) {
        if (graph == null) {
                return;
        }

        // do something with graph
}
```

runSomeAlgorithm.java hosted with ❤ by GitHub view raw

```java
private static void runSomeAlgorithm(Graph graph) {

        // do something with graph
}
```

runSomeAlgorithmOptimized.java hosted with ❤ by GitHub view raw

Branch Prediction

- Find hotter code

```java
private static int isOpt(int x, int y) {
        int veryHardCalculation = 0;

        if (x >= y) {
                veryHardCalculation = x * 1000 + y;
        } else {
                veryHardCalculation = y * 1000 + x;
        }
        return veryHardCalculation;
}
```

isOpt1.java hosted with ❤ by GitHub view raw

```java
private static int isOpt(int x, int y) {
        int veryHardCalculation = 0;

        if (x < y) {
                // this would not require a jump
                veryHardCalculation = y * 1000 + x;
                return veryHardCalculation;
        } else {
                veryHardCalculation = x * 1000 + y;
                return veryHardCalculation;
        }
}
```

isOpt2.java hosted with ❤ by GitHub view raw

Loop Unrolling

```java
private static double[] loopUnrolling(double[][] matrix1, double[] vector1) {
        double[] result = new double[vector1.length];

        for (int i = 0; i < matrix1.length; i++) {
        for (int j = 0; j < vector1.length; j++) {
                        result[i] += matrix1[i][j] * vector1[j];
                }
        }

        return result;
}
```

rolledLoop.java hosted with ❤ by GitHub view raw

```java
private static double[] loopUnrolling2(double[][] matrix1, double[] vector1) {
        double[] result = new double[vector1.length];

        for (int i = 0; i < matrix1.length; i++) {
                result[i] += matrix1[i][0] * vector1[0];
                result[i] += matrix1[i][1] * vector1[1];
                result[i] += matrix1[i][2] * vector1[2];
                // and maybe it will expand even further - e.g. 4 iterations, thu
                // adding code to fix the indexing
                // which we would waste more time doing correctly and efficiently
        }

        return result;
}
```

Control Flow Optimizations

- Control flow optimizations analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency.

Code Optimizer Example

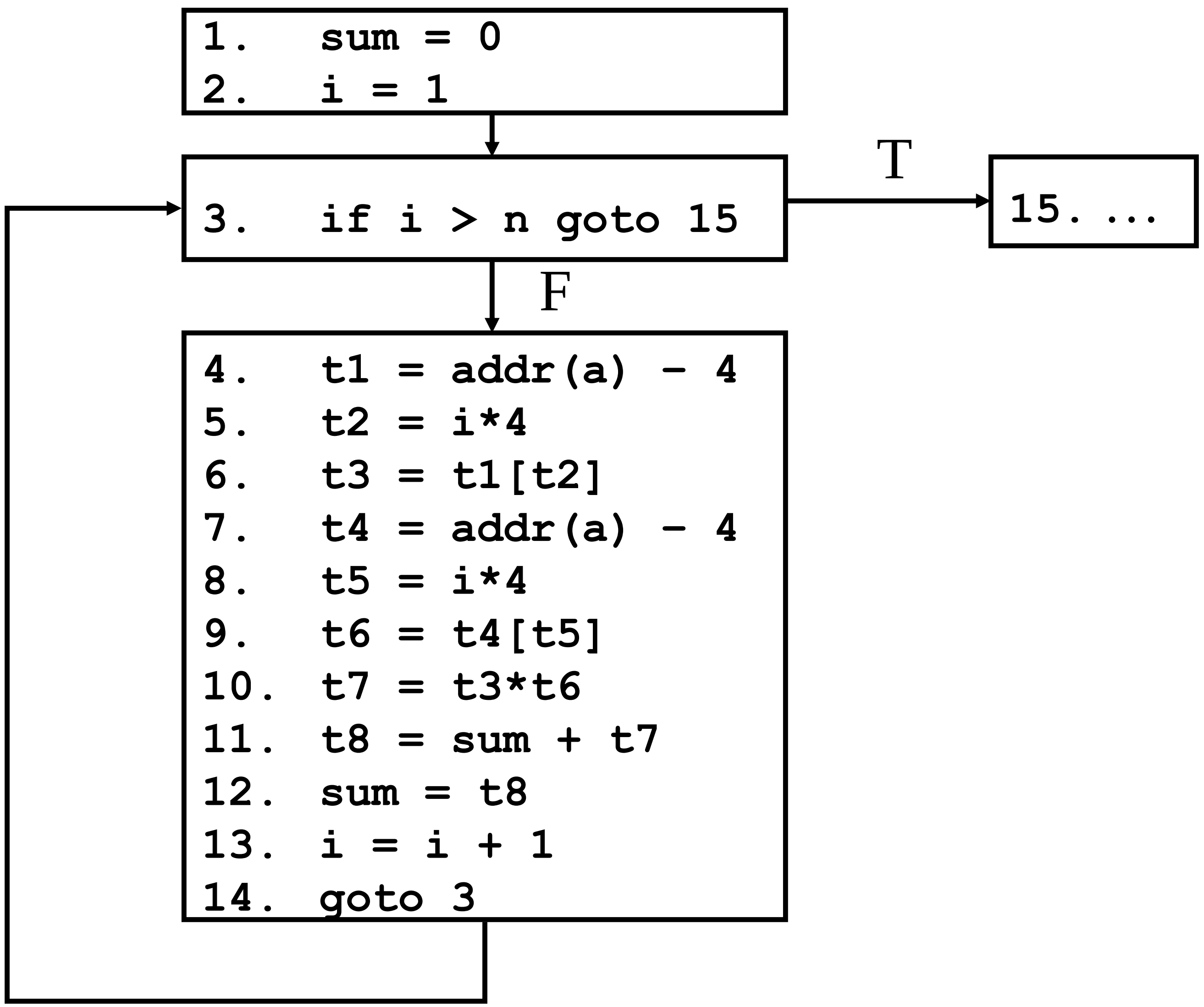
- Define classical optimizations using an example Fortran loop
- Opportunities result from table-driven code generation

```
 …
 sum = 0
 do 10 i = 1, n
10  sum = sum + a[i]*a[i]
 …
```

Three Address Code

```
1.   sum = 0                  initialize sum
2.   i = 1                    initialize loop counter
3.   if i > n goto 15         loop test, check for limit
4.   t1 = addr(a) - 4
5.   t2 = i * 4               a[i]
6.   t3 = t1[t2]
7.   t4 = addr(a) - 4
8.   t5 = i * 4               a[i]
9.   t6 = t4[t5]
10.  t7 = t3 * t6             a[i]*a[i]
11.  t8 = sum + t7
12.  sum = t8                 increment sum
13.  i = i + 1                increment loop counter
14.  goto 3

15.  …
```

Control Flow Graph (CFG)



Common Subexpression Elimination

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15          3.
4.   t1 = addr(a) - 4          4.
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) - 4          7.
8.   t5 = i*4
9.   t6 = t4[t5]
10.  t7 = t3*t6
11.  t8 = sum + t7
12.  sum = t8
13.  i = i + 1                 11a
14.  goto 3
15.  …
```

Common Subexpression Elimination

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i*4
6.   t3 = t1[t2]
7.   t4 = addr(a) - 4
8.   t5 = i*4
9.   t6 = t4[t5]
10.  t7 = t3*t6
11.  t8 = sum + t7
12.  sum = t8
13.  i = i + 1
14.  goto 3
15.  …
```

```
       1.  sum = 0
       2.  i = 1
3.  if i > n goto 15
4.  t1 = addr(a) - 4
       5.  t2 = i*4
       6.  t3 = t1[t2]
7.  t4 = addr(a) - 4
       8.  t5 = i*4
       9.  t6 = t4[t5]
       10. t7 = t3*t6
       10a t7 = t3*t3
       11. t8 = sum + t7
11a sum = sum + t7
       12. sum = t8
       13. i = i + 1
       14. goto 3
```

Invariant Code Motion

```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
```

```
1.   sum = 0
2.   i = 1
2a   t1 = addr(a) - 4
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
```

Invariant Code Motion

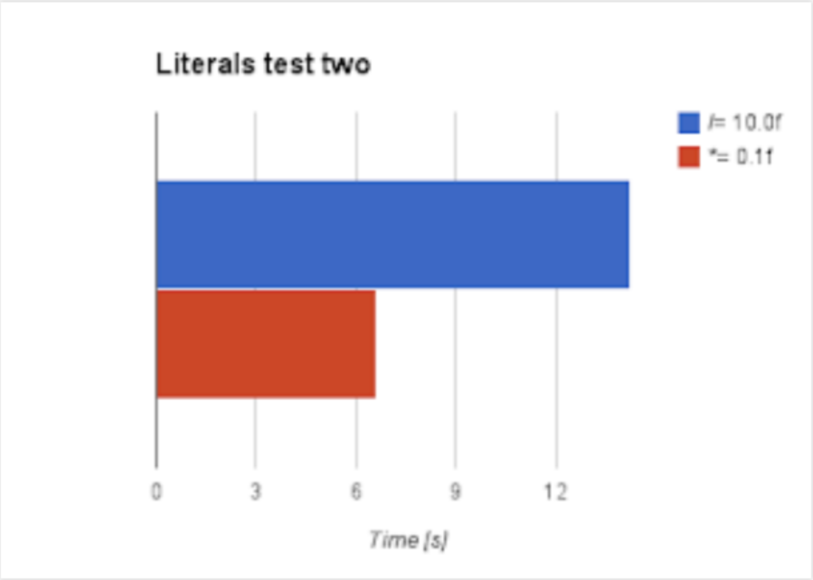
```
1.   sum = 0
2.   i = 1
3.   if i > n goto 15
4.   t1 = addr(a) - 4
5.   t2 = i * 4
6.   t3 = t1[t2]
10a t7 = t3 * t3
11a sum = sum + t7
13. i = i + 1
14. goto 3
15. …
```

Native Code Generation

- Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions; some small optimizations are performed according to architecture characteristics.
  - E.g., SSE4 (SSE4.1 and SSE4.2)
    - SSE4.1: 47
    - SSE4.2: 47 + 7

Performance

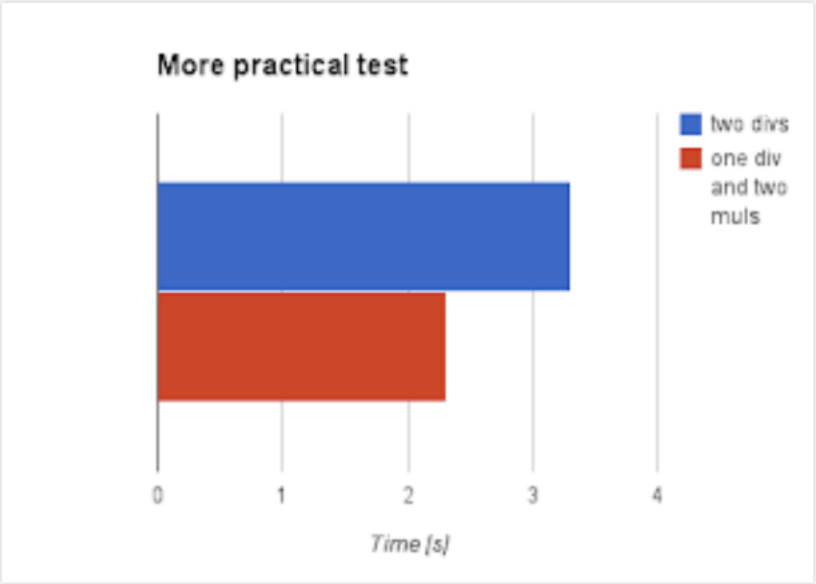
- Division vs. Multiplication




```
a *= 0.1f;
```

vs

```
a /= 10.0f;
```




```
sum += a / r;
sum += b / r;
```

vs

```
float den = 1 / r;
sum += a * den;
sum += b * den;
```

메모리 관리

메모리 관리 기본

- 다 쓴 객체 참조를 해제하라

```java
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() throws Exception {
        if (size == 0)
            throw new Exception();
        return elements[--size];
    }
```

```java
public Object pop() throws Exception{
    if (size == 0)
        throw new Exception();
    Object result = elements[--size];
    elements[size] = null; // 다 쓴 참조 해제
    return result;
}
```

Garbage Collection

- Invented in 1959
- Automatic memory management
  - The GC reclaims memory occupied by objects that are no longer in use
  - Such objects are called garbage
- Conceptually simple
  - Scan objects in memory, identify objects that cannot be accessed (now, or in the future)
  - Reclaim these garbage objects
- In practice, very tricky to implement

Manual Reference Counting

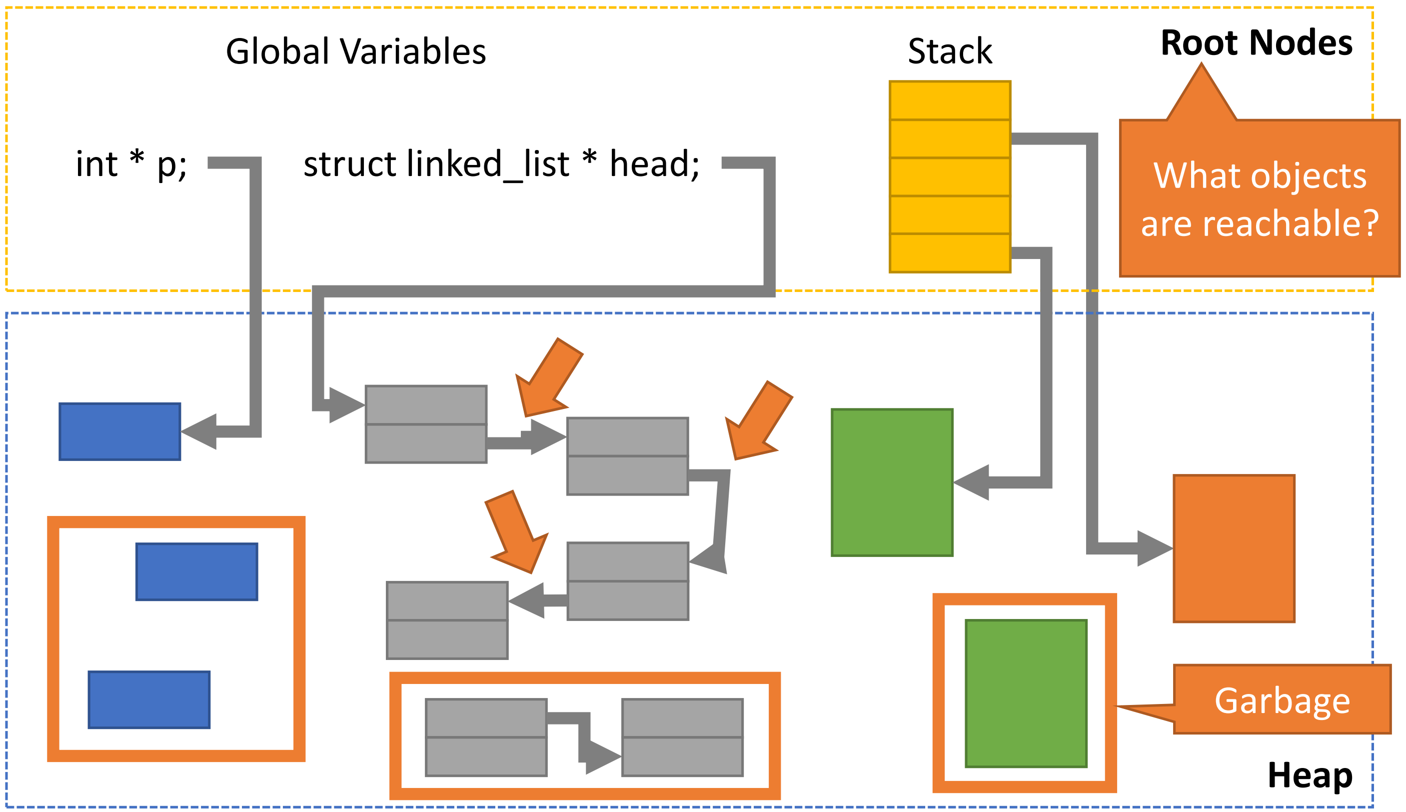
- Idea: keep track of how many references there are to each object in a reference counter stored with each object
  - Copy a reference to an object globalvar = q
    - increment count: “addref”
  - Remove a reference p = NULL
    - decrement count: “release”
- Uses set of rules programmers must follow
  - E.g., must ‘release’ reference obtained from OUT parameter in function call
  - Must ‘addref’ when storing into global
  - May not have to use addref/release for references copied within one function
- Programmer must use addref/release correctly
  - Still somewhat error prone, but rules are such that correctness of the code can be established locally without consulting the API documentation of any functions being called; parameter annotations (IN, INOUT, OUT, return value) imply reference counting rules

Automatic Reference Counting

- Idea: *force* automatic reference count updates when pointers are assigned/copied
- Most common variant:
  - C++ allows programmer to interpose on assignments and copies via operator overloading/special purpose constructors.

- Disadvantage of all reference counting schemes is their inability to handle cycles
  - But great advantage is immediate reclamation: no “drag” between last access & reclamation

Garbage Collection Concepts



Approaches to GC

- **Reference Counting**
  - Each object keeps a count of references
  - If an objects count == 0, it is garbage
- **Mark and Sweep**
  - Starting at the roots, traverse objects and “mark” them
  - Free all unmarked objects on the heap
- **Copy Collection**
  - Extends mark & sweep with compaction
  - Addresses CPU and external fragmentation issues
- **Generational Collection**
  - Uses heuristics to improve the runtime of mark & sweep

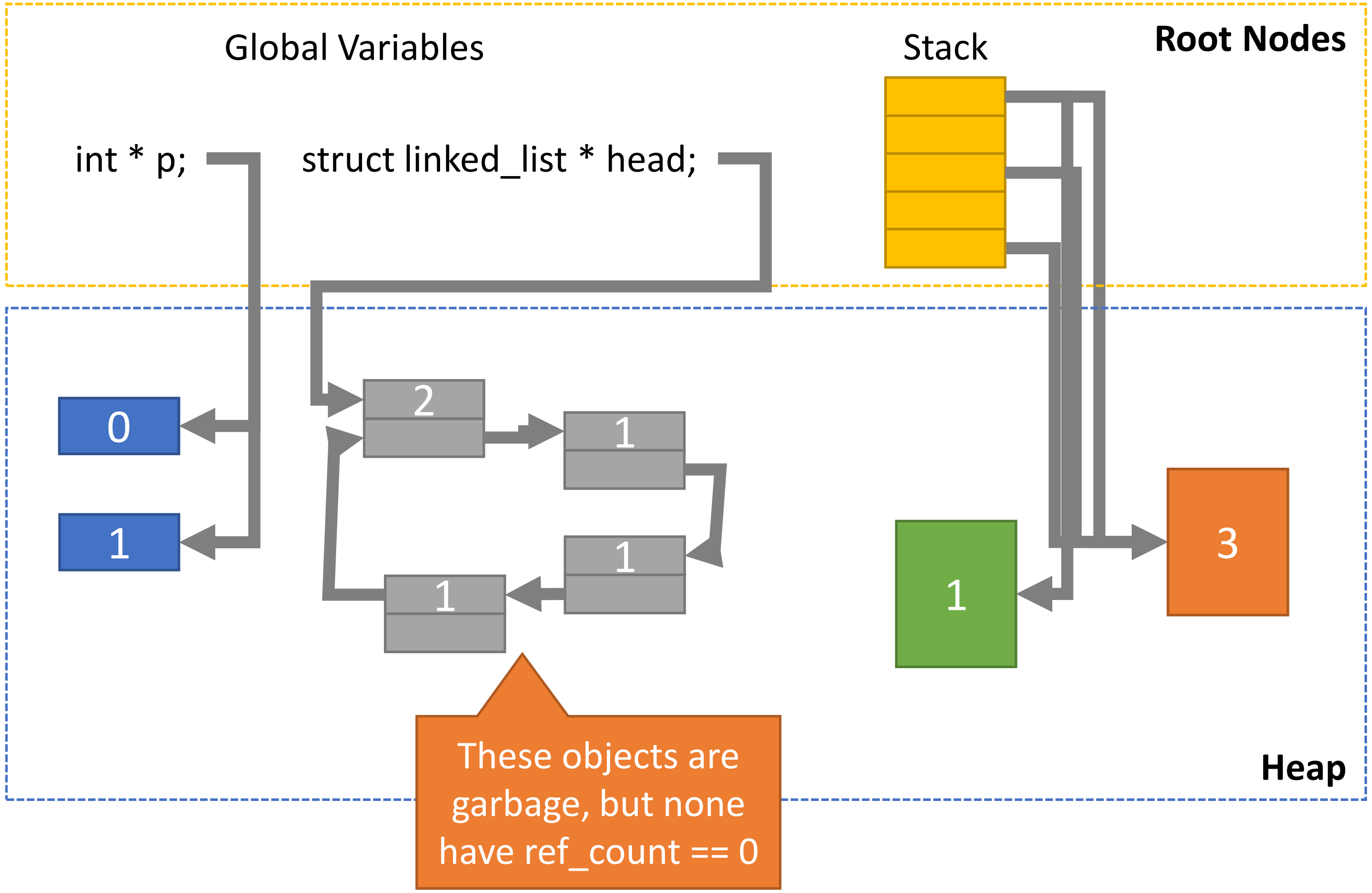
Reference Counting

- Key idea: each object includes a ref_count

```
Assume obj * p = NULL;
p = obj1; // obj1->ref_count++
p = obj2; // obj1->ref_count--, obj2->ref_count++
```

- If an object’s ref_count == 0, it is garbage
  - No pointers target that object
  - Thus, it can be safely freed

Reference Counting Example



Pros and Cons of Reference Counting

The Good

- Relatively easy to implement
- Easy to conceptualize

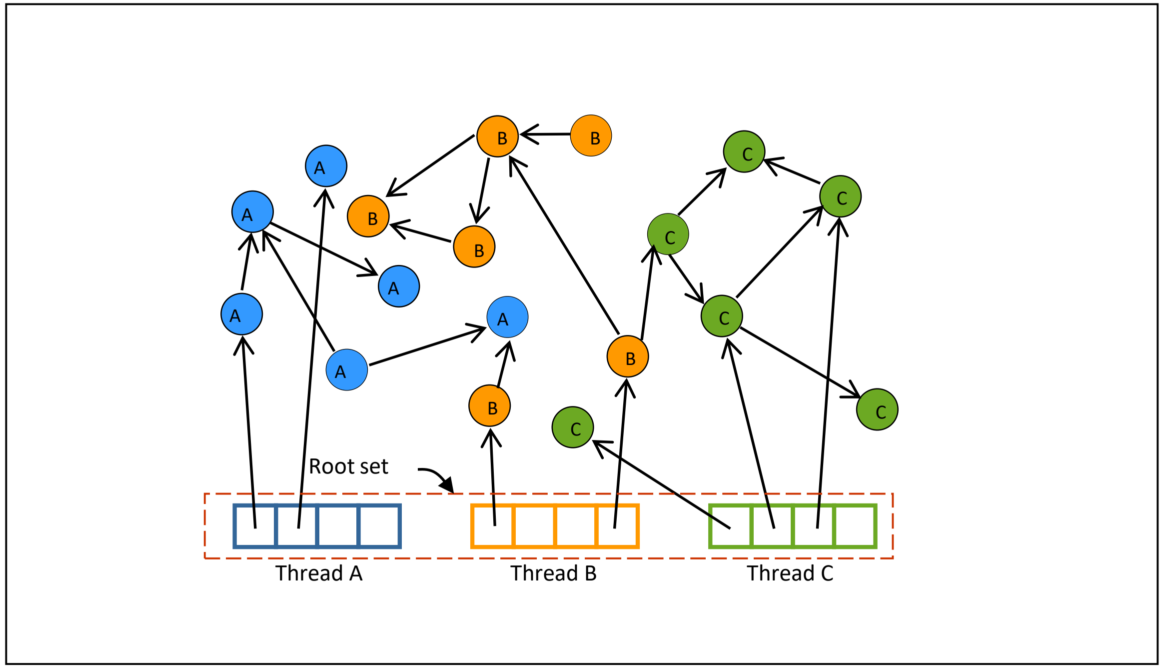
The Bad

- Not guaranteed to free all garbage objects
- Additional overhead (int ref_count) on all objects
- Access to obj->ref_count must be synchronized

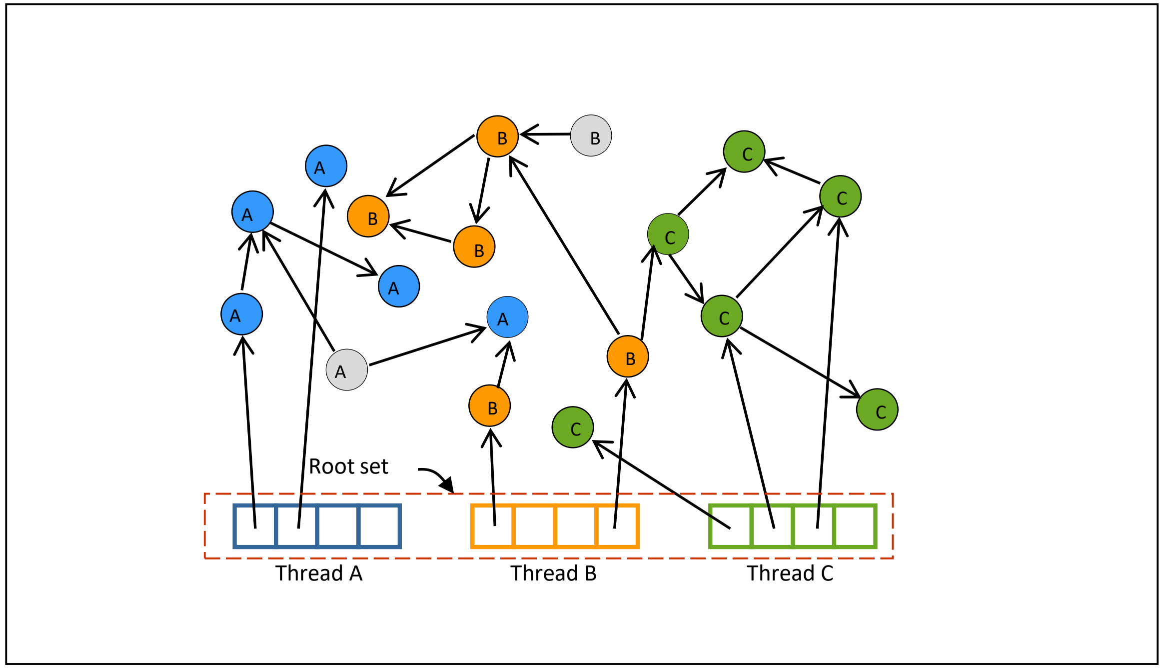
Mark and Sweep

- Key idea: periodically scan all objects for reachability
  - Start at the roots
  - Traverse all reachable objects, mark them
  - All unmarked objects are garbage

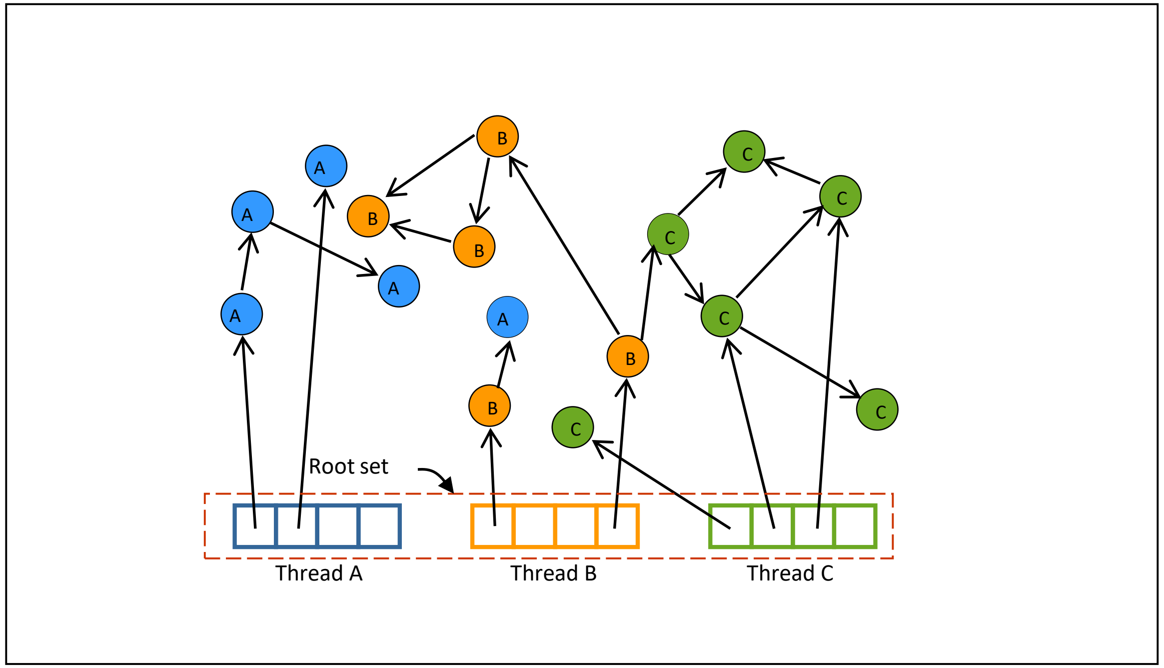
Reachability Graph



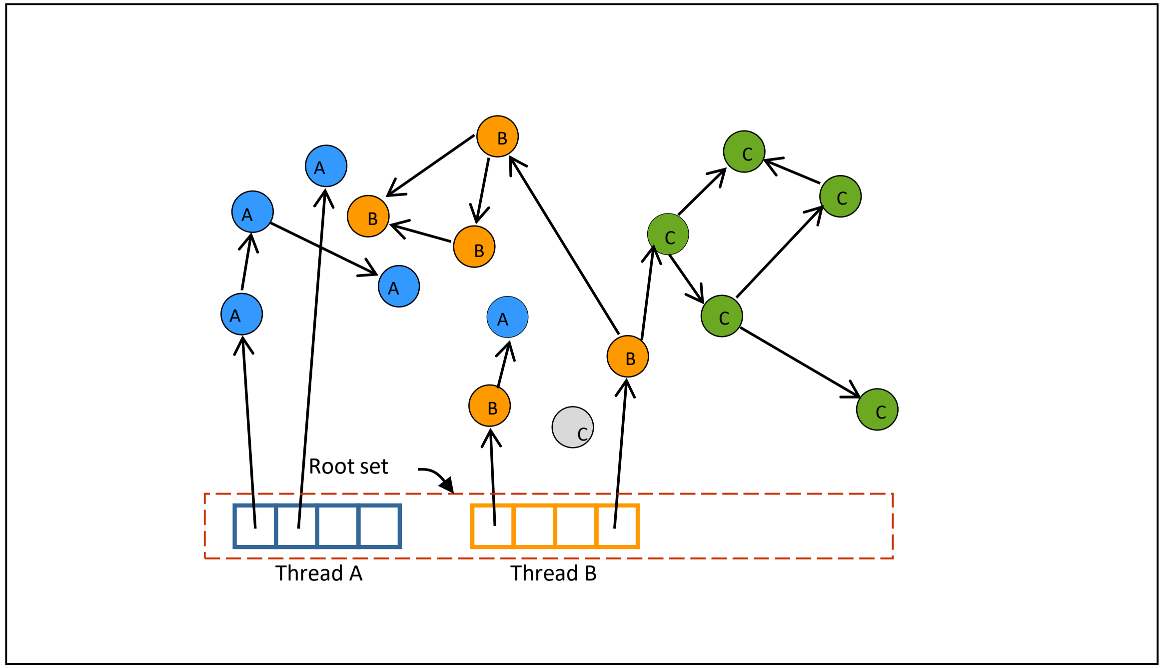
Reachability Graph



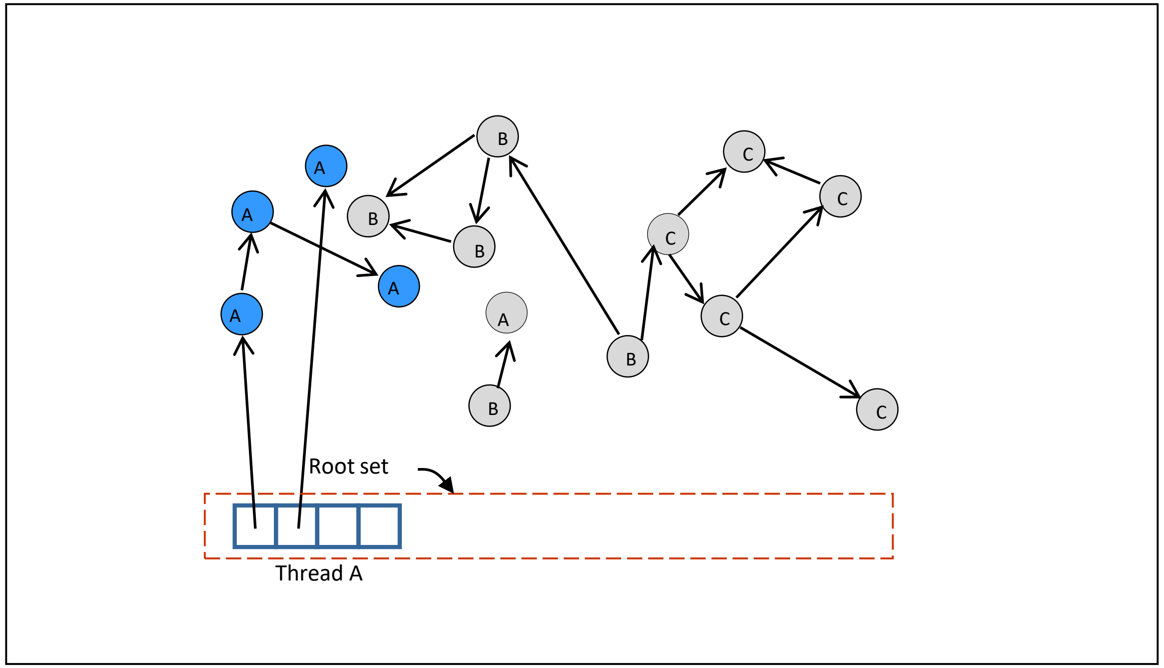
Reachability Graph



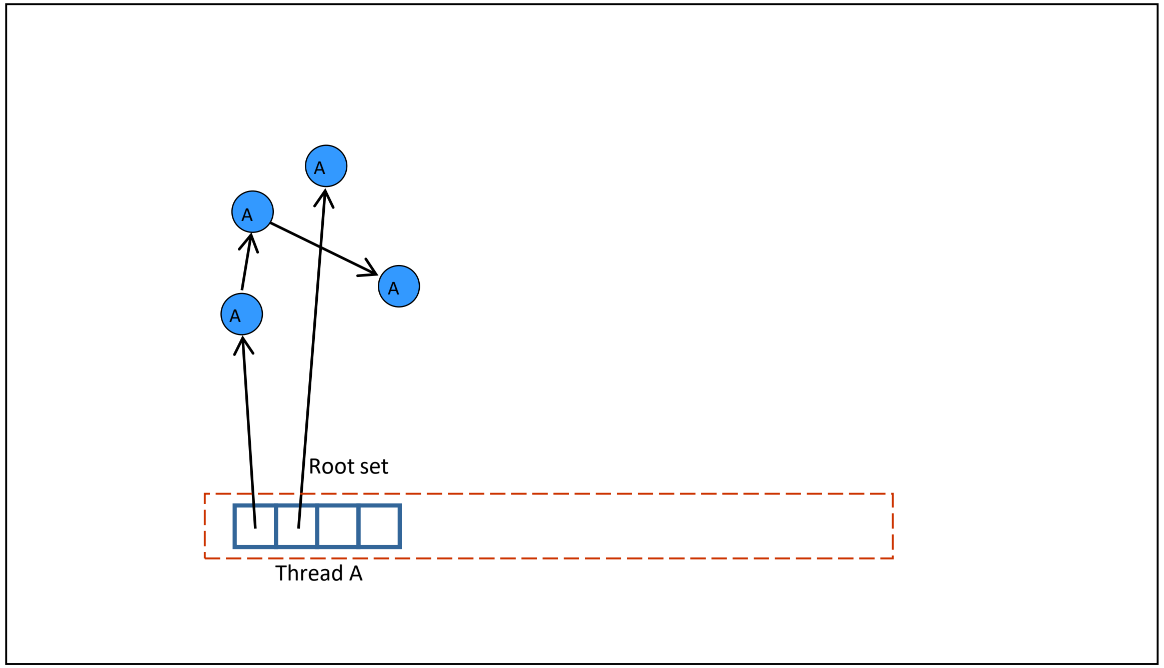
Reachability Graph



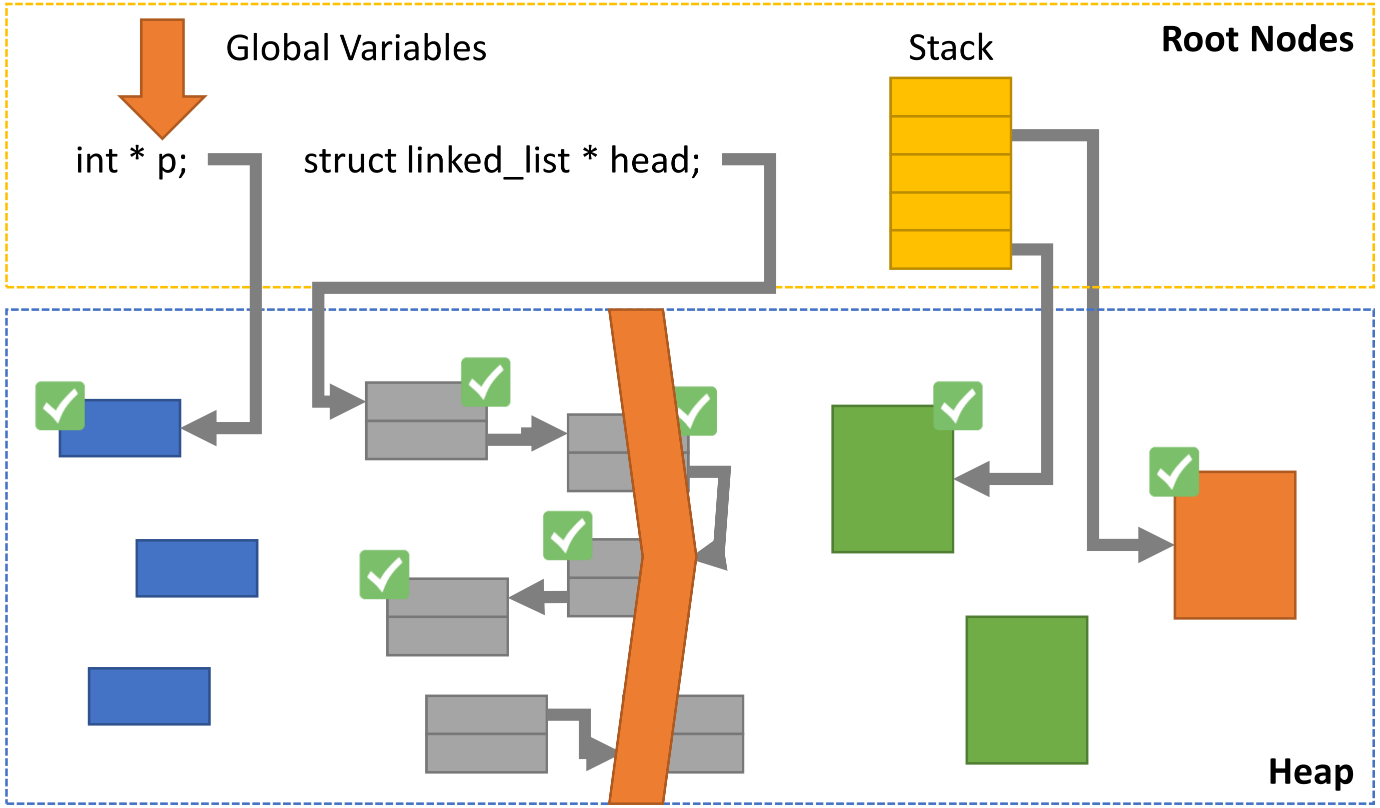
Reachability Graph



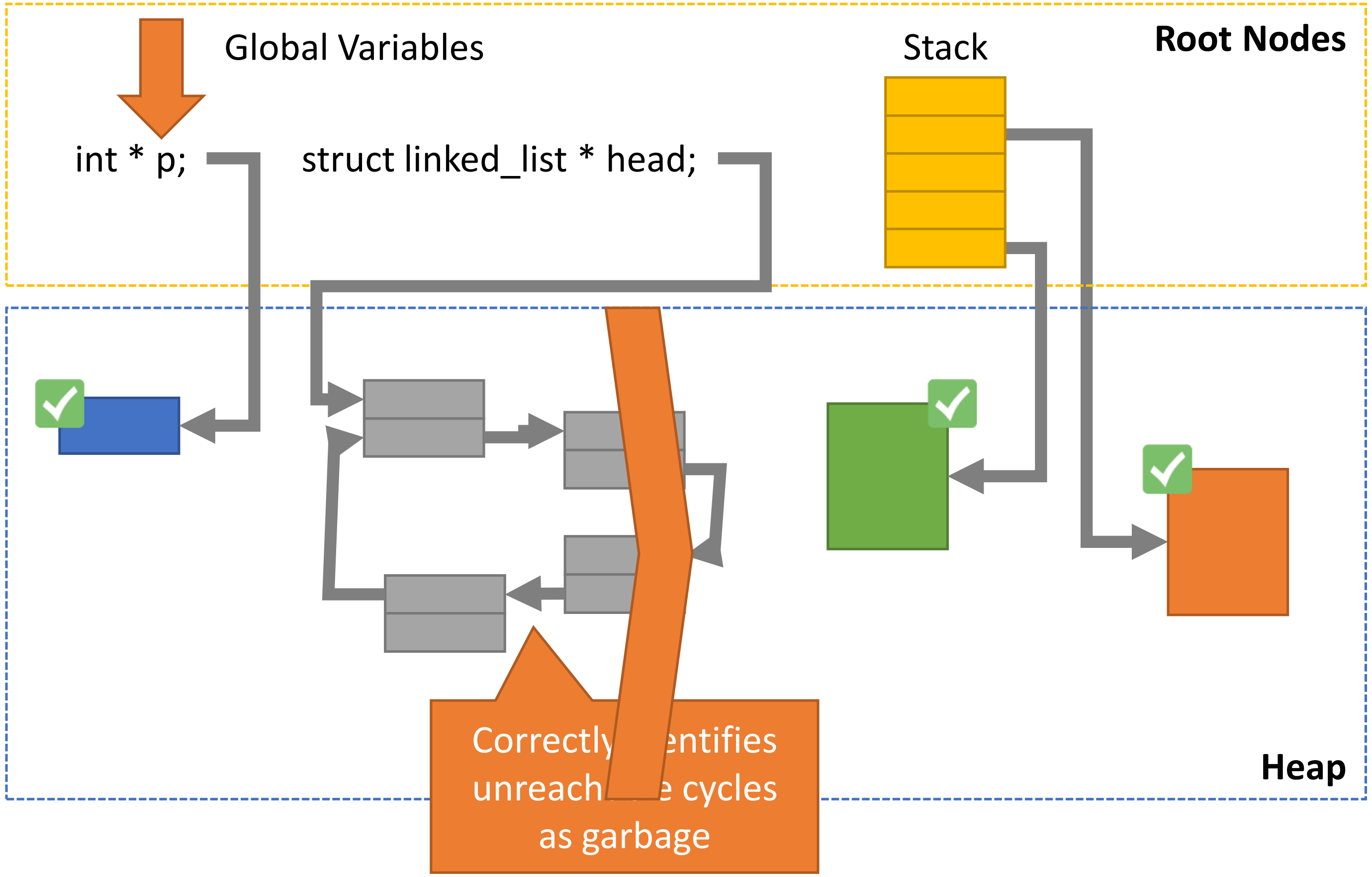
Reachability Graph



Mark and Sweep Example



Mark and Sweep Example



Pros and Cons of Mark and Sweep

The Good

- Overcomes the weakness of reference counting
- Fairly easy to implement and conceptualize
- Guaranteed to free all garbage objects

Be careful: if you forget to set a reference to NULL, it will never be collected (i.e. Java can leak memory)

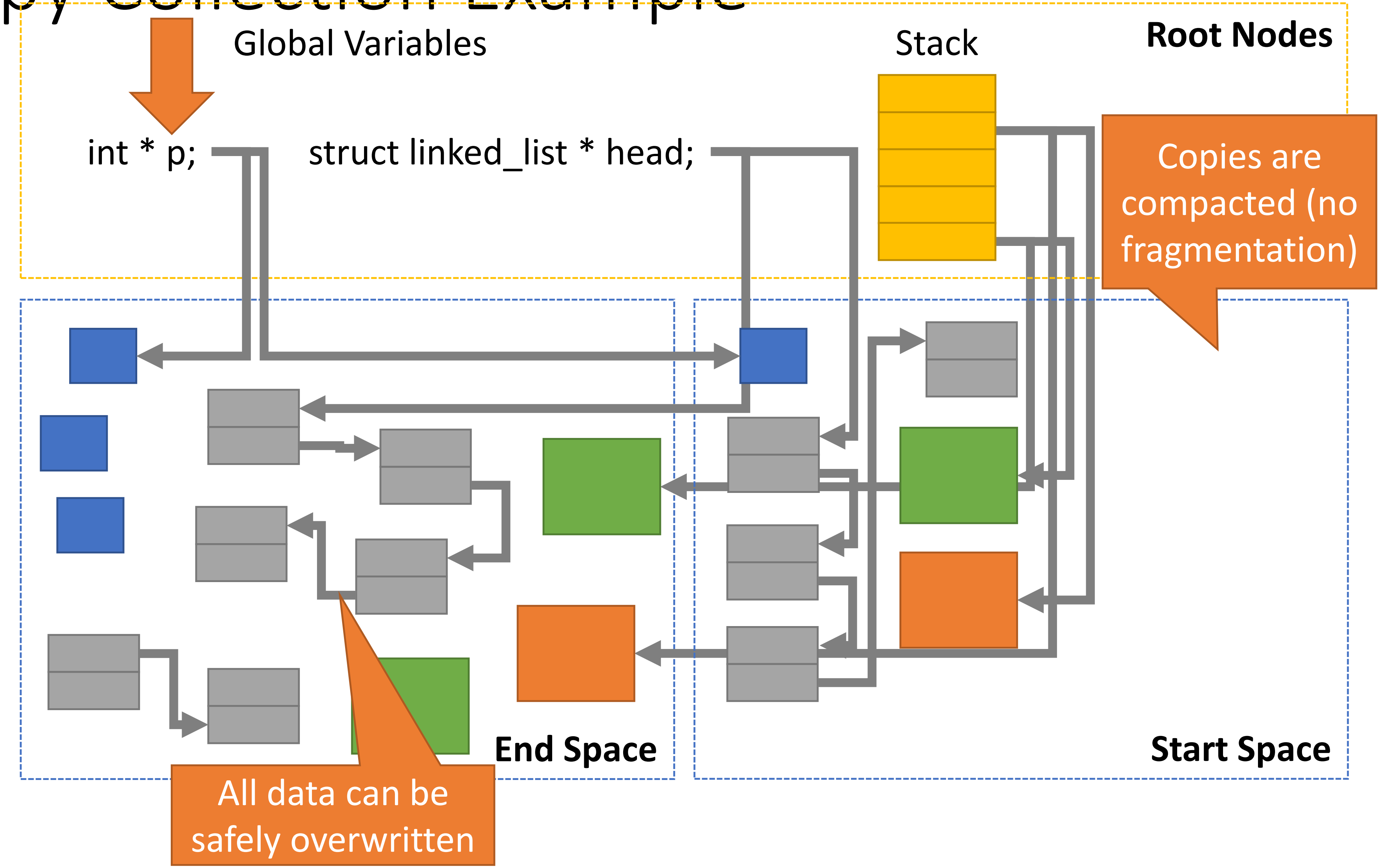
The Bad

- Mark and sweep is CPU intensive
  - Traverses all objects reachable from the root
  - Scans all objects in memory freeing unmarked objects
- Naïve implementations “stop the world” before collecting
  - Threads cannot run in parallel with the GC
  - All threads get stopped while the GC runs

Copy Collection

- Problem with mark and sweep:
  - After marking, all objects on the heap must be scanned to identify and free unmarked objects
- Key idea
  - Divide the heap into *start space* and *end space*
  - Objects are allocated in *start space*
  - During GC, instead of marking, copy live object from *start space* into *end space*
  - Switch the *space* labels and continue

Copy Collection Example



Pros and Cons of Copy Collection

The Good

- Improves on mark and sweep
- No need to scan memory for garbage to free
- After compaction, there is no fragmentation

The Bad

- Copy collection is slow
  - Data must be copied
  - Pointers must be updated
- Naïve implementations are not parallelizable
  - “Stop the world” collector

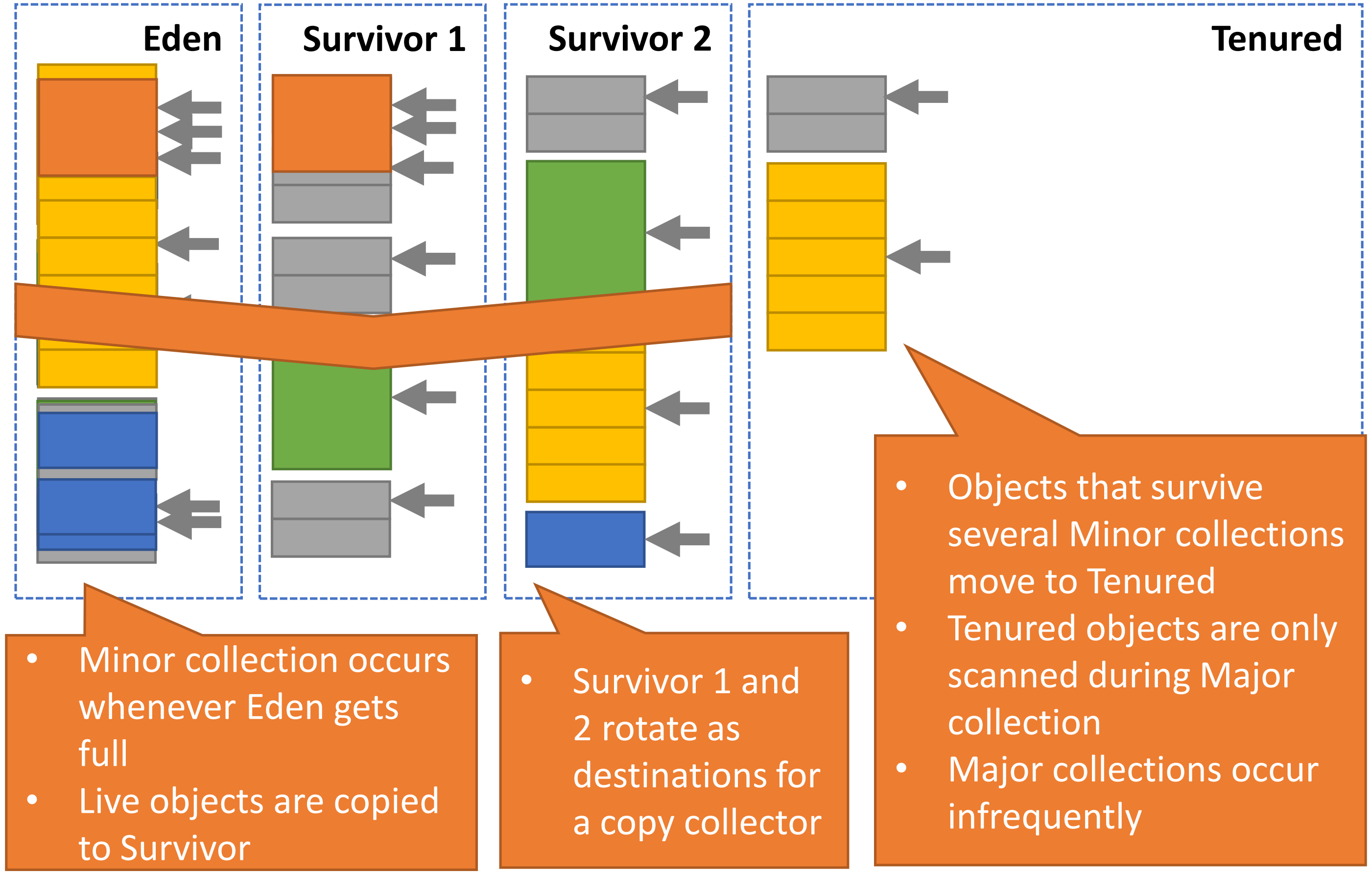
Generational Collection

- Problem: mark and sweep is slow
  - Expensive full traversals of live objects
  - Expensive scan of heap memory
- Problem: copy collection is also slow
  - Expensive full traversals of live objects
  - Periodically, all live objects get copied
- Solution: leverage knowledge about object creation patterns
  - Object lifetime tends to be inversely correlated with likelihood of becoming garbage (generational hypothesis)
  - Young objects die quickly – old objects continue to live

Garbage Collection in Java

- By default, most JVMs use a generational collector
- GC periodically runs two different collections:
  - Minor collection – occurs frequently
  - Major collection – occurs infrequently
- Divides heap into 4 regions
  - Eden: newly allocated objects
  - Survivor 1 and 2: objects from Eden that survive minor collection
  - Tenured: objects from Survivor that survive several minor collections

Generational Collection Example



malloc()/free() vs. GC

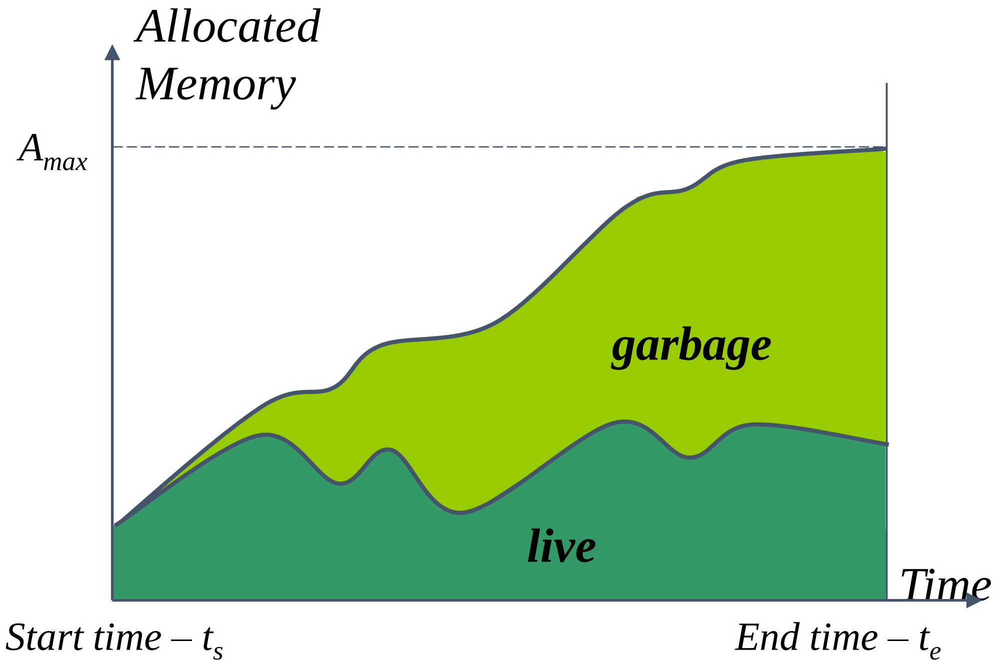
Explicit Alloc/Dealloc

- Advantages:
  - Typically faster than GC
  - No GC “pauses” in execution
  - More efficient use of memory
- Disadvantages:
  - More complex for programmers
  - Tricky memory bugs
    - Dangling pointers
    - Double-free
    - Memory leaks
  - Bugs may lead to security vulnerabilities

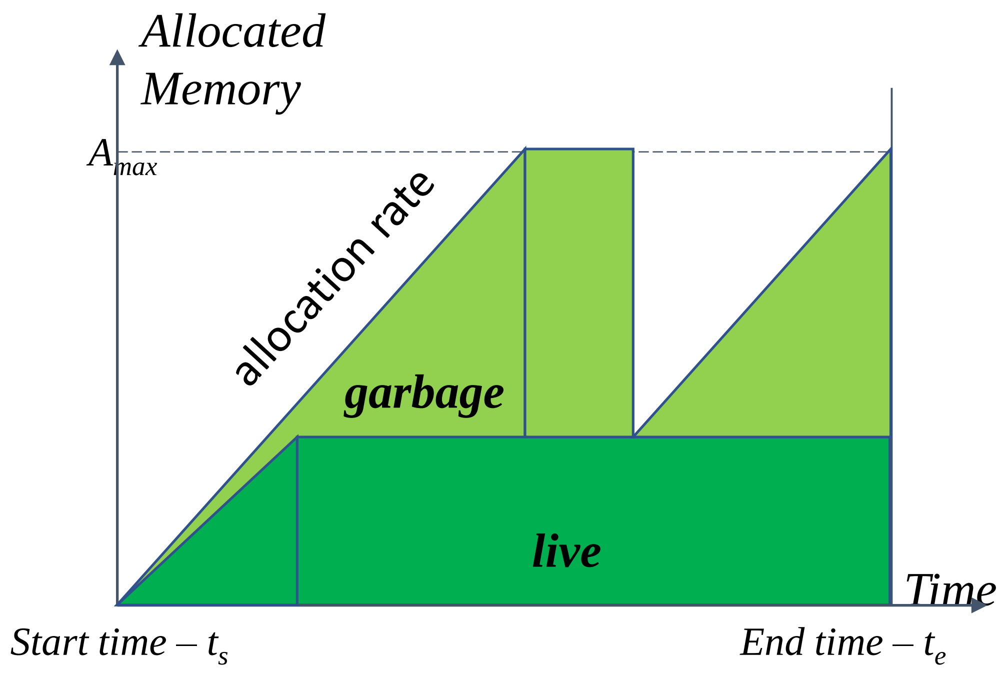
Garbage Collection

- Advantages:
  - Much easier for programmers
- Disadvantages
  - Typically slower than explicit alloc/dealloc
  - Good performance requires careful tuning of the GC
  - Less efficient use of memory
  - Complex runtimes may have security vulnerabilities
    - JVM gets exploited all the time

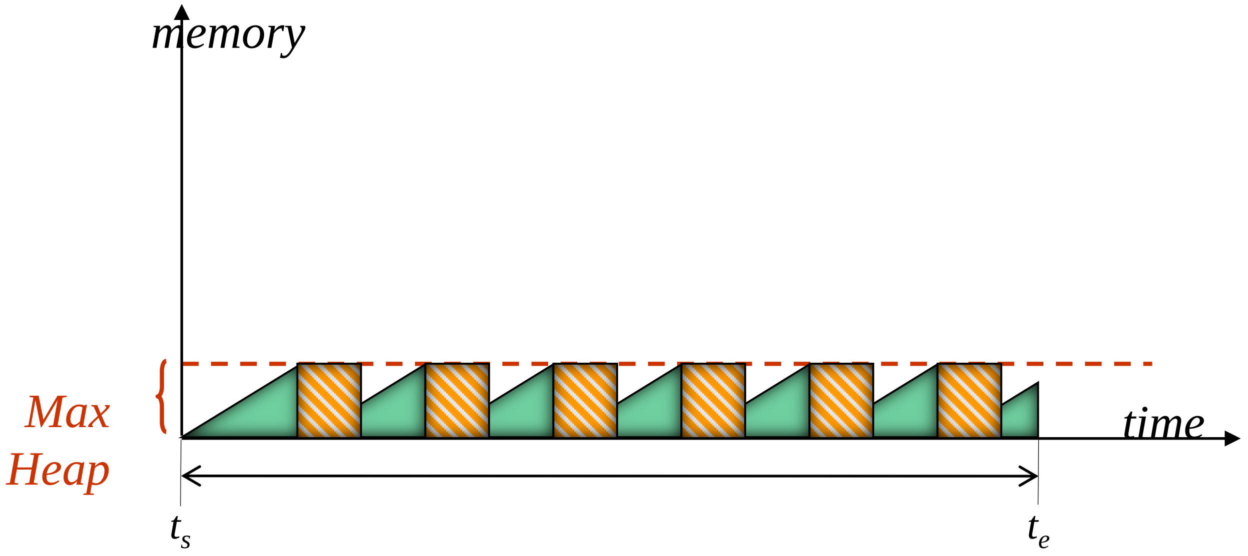
Memory Allocation Time-Profile



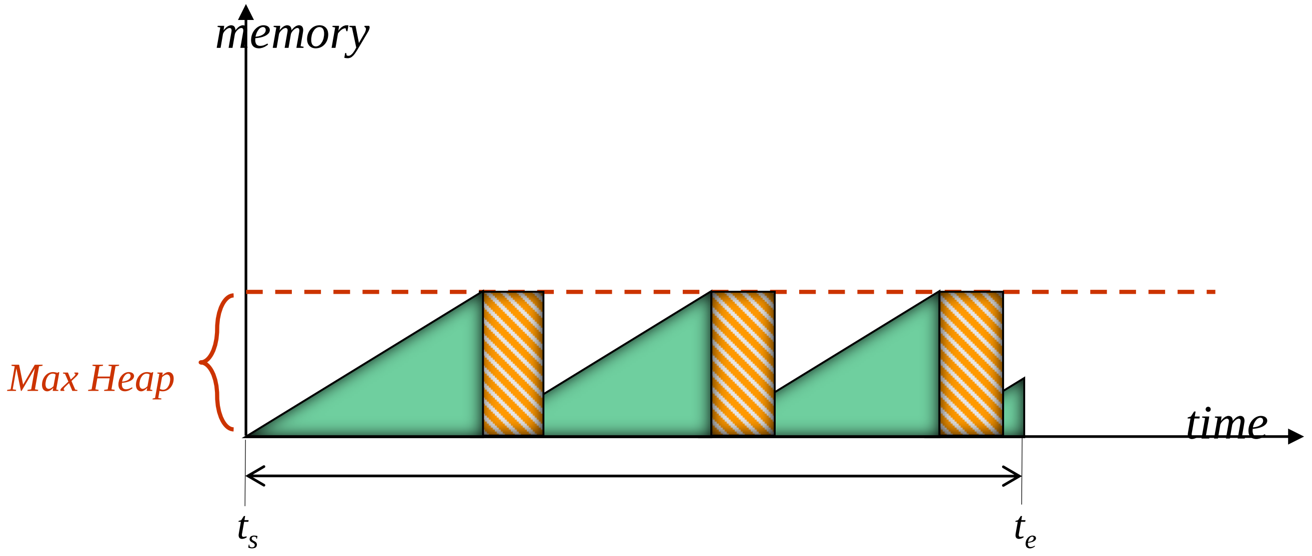
Modeling Memory Allocation



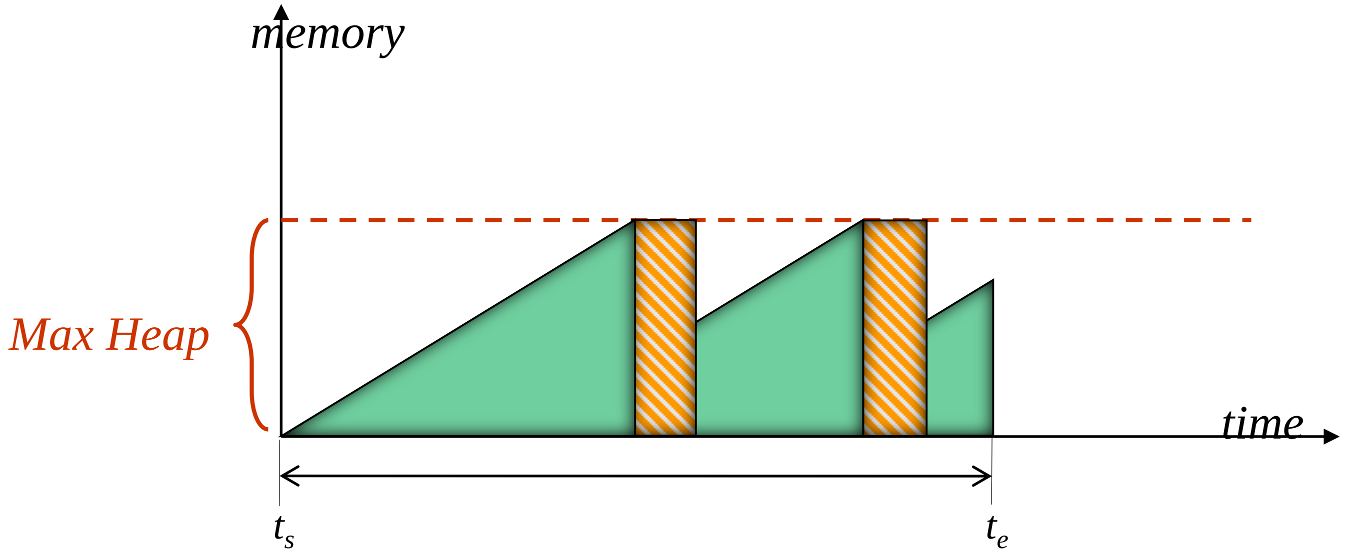
Execution Time vs. Memory



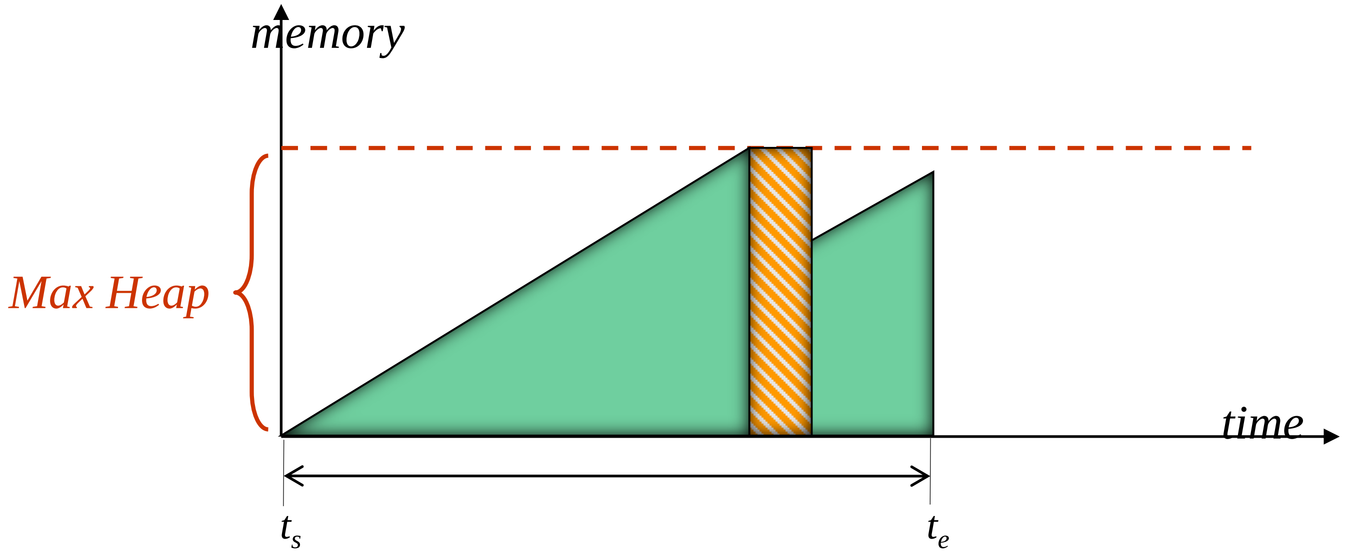
Execution Time vs. Memory



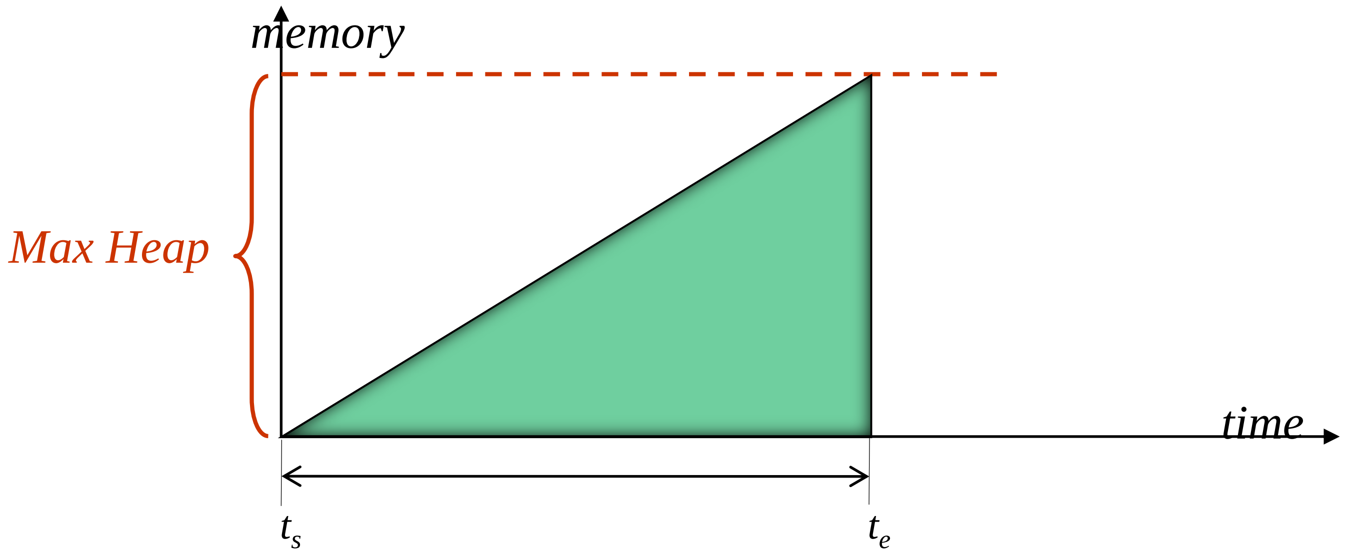
Execution Time vs. Memory



Execution Time vs. Memory



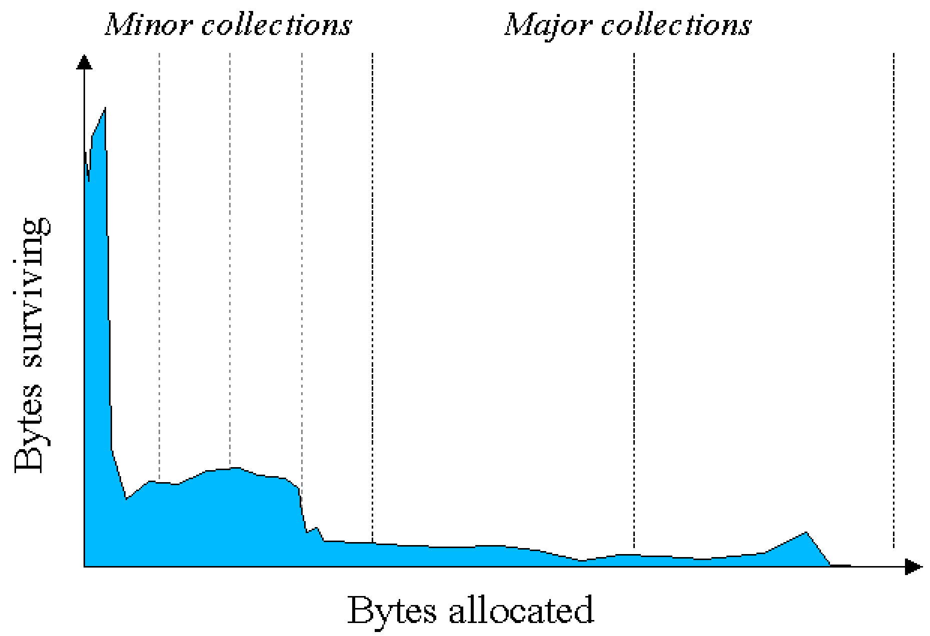
Execution Time vs. Memory



Heap Size vs. GC Frequency

- All else being equal, smaller maximum heap sizes necessitate more frequent collections
  - Old rule of thumb: need between 1.5x and 2.5x times the size of the live heap to limit collection overhead to 5-15% for applications with reasonable allocation rates
  - [Hertz 2005] finds that GC outperforms explicit MM when given 5x memory, is 17% slower with 3x, and 70% slower with 2x
  - Performance degradation occurs when live heap size approaches maximum heap size

Infant Mortality



Source: http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html

Generational Collection

- Observation: “most objects die young”
- Allocate objects in separate area (“nursery”, “Eden space”), collect area when run out of space
  - Will typically have to evacuate few survivors
  - “minor garbage collection”
- But: must treat all pointers into Eden as roots
  - Typically, requires cooperation of the mutator threads to record assignments: if ‘b’ is young, and ‘a’ is old, a.x = b must add a root for ‘b’.
    - Aka “write barrier”

More DS…

- 이진 검색 트리
- 구간 트리
- 트라이

- 깊이 우선 탐색
- 너비 우선 탐색
- 최단 경로 알고리즘
- 최소 스패닝 트리