Comparisons

Persistent Array
Persistent LinkedList
Persistent Stack
Persistent Queue
Persistent Search Tree

We have done BenchMarking using: GoogleBenchMark Tool
https://github.com/google/benchmark

PROJECT MEMBERS: ANANNYO DEY, SOUMYAJIT RUDRA SARMA , DEBASMIT ROY, KANKO GHOSH AND KUSHAL DAS

Persistent Array

Persistent Array

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Array

Strategies	Update(index, version, newVal)	RetrieveData(index, version)
Copy_On_Write	O(length_of_array_at_version) ~ O (N)	O(1)
Fat_Node	**O(1)**	**Between O(1) and O(V)[Worst Case] V = number of versions O(1) (Average)**
Log_log_time method (Deitz, 1989)	**O(log(log(min(n, v))), n = size of array v = number of versions**	**O(log(log(min(n, v))), n = size of array v = number of versions**

Persistent Array

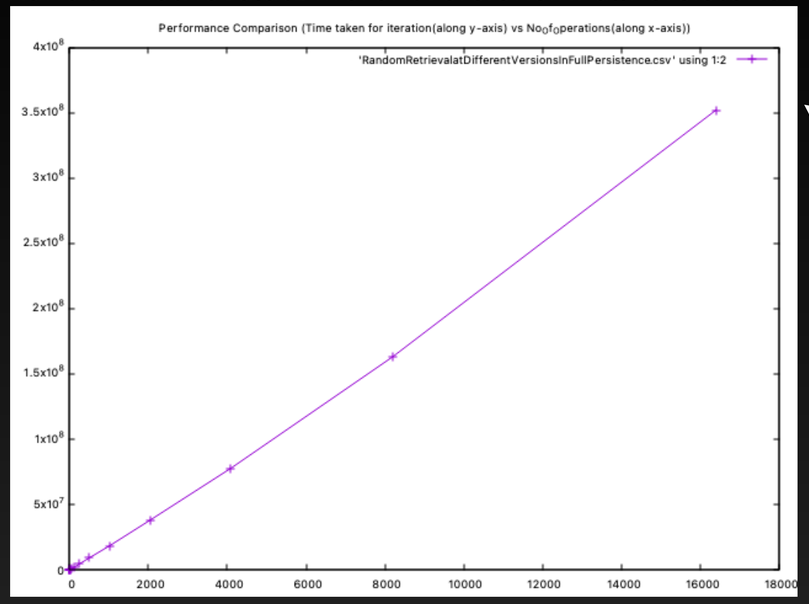
Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Array

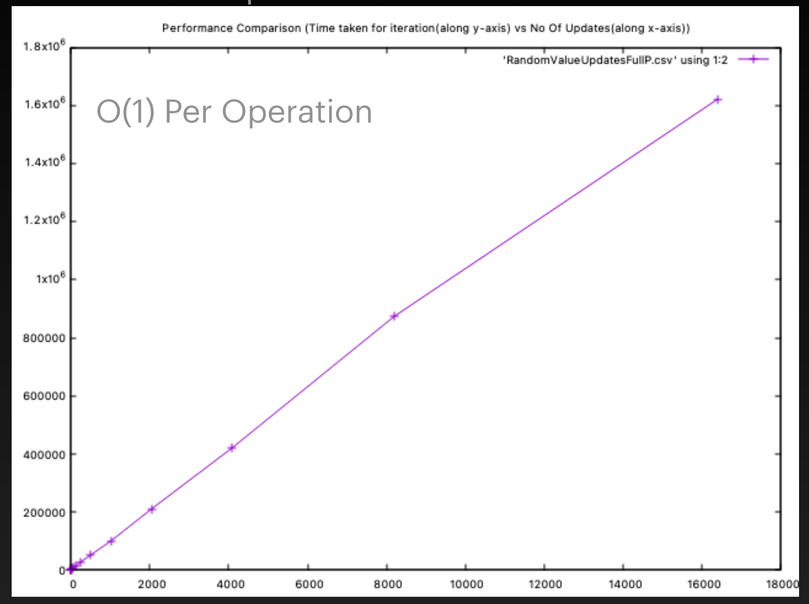
Strategies	Category1	Category2
Copy_On_Write	~ O (N^2) To Hold All The Copies of Array At Different Versions	O(V) To Hold The Mapping From Version To Array
Fat_Node	**O(N + V)** **N = size of initial array,** **V = number of version**	**O(V)** **To Hold The Mapping** **From Version To immediate ancestor version**
Log_log_time method (Deitz, 1989)	**O(N + V)** **N = size of initial array,** **V = number of version**	**O(V)** **to arrange the versions in form of list order maintenance tree**

Benchmarking Of FatNode Model

RandomRetrievalatDifferentVersionsinFullPersistence

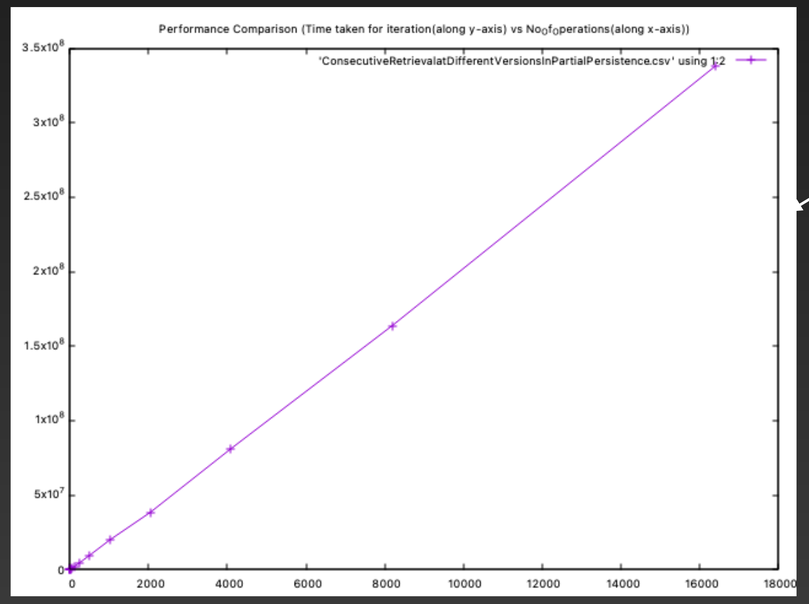


RandomValueUpdatesFullPersistence

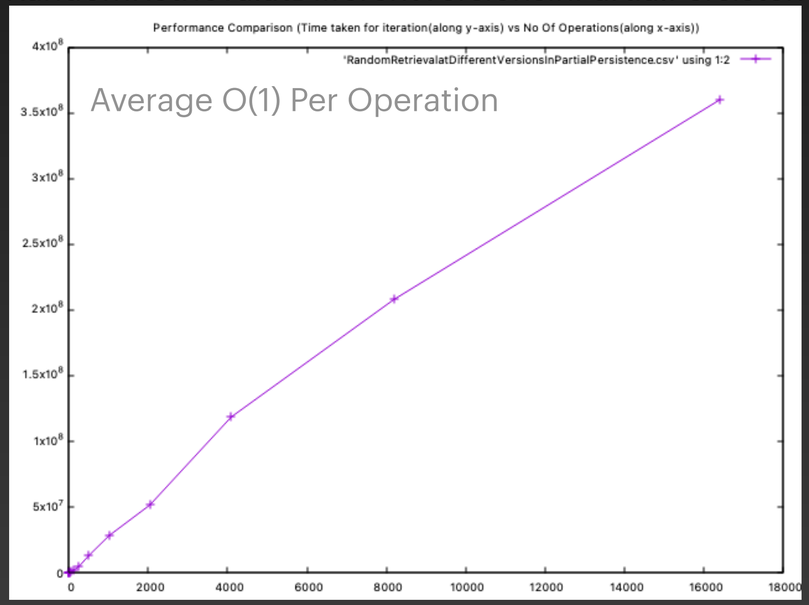


Average O(1) Per Operation

ConsecutiveRetrievalatDifferentVersionsinPartialPersistence



RandomRetrievalatDifferentVersionsinPartialPersistence



Partial Persistent Linked List

Partial Persistent Linked List

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions,
m = no. of modifications in a particular position

Strategies	InsertAfter(position) #	DeleteAfter(position) #	UpdateData (position) #	RetrieveData (postition,version)	traverseWholeLL atVer(version)
Fat Node	O(1)	O(1)	O(1)	O(log_x m) [If the versions are stored in a Balanced x_array Tree/ Trie] O(1) [Amortised]	O(N * log_x m)
Path Copying	O(1) for at extreme position O(N) In Average	O(1) for at extreme position O(N) for at Rear	O(1) for at extreme position O(N) for at Rear	O(N)	O(N)
Pointer Machine	O(1) [Amortised]	O(1) [Amortised]	O(1) [Amortised]	O(N)	O(N)
Ephemeral Linked List	O(1)	O(1)	O(1)	O(N) [Only Current Version Supported]	O(N) [Only Current Version Supported]

time to reach the node at that position is not considered

Partial Persistent Linked List

Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions

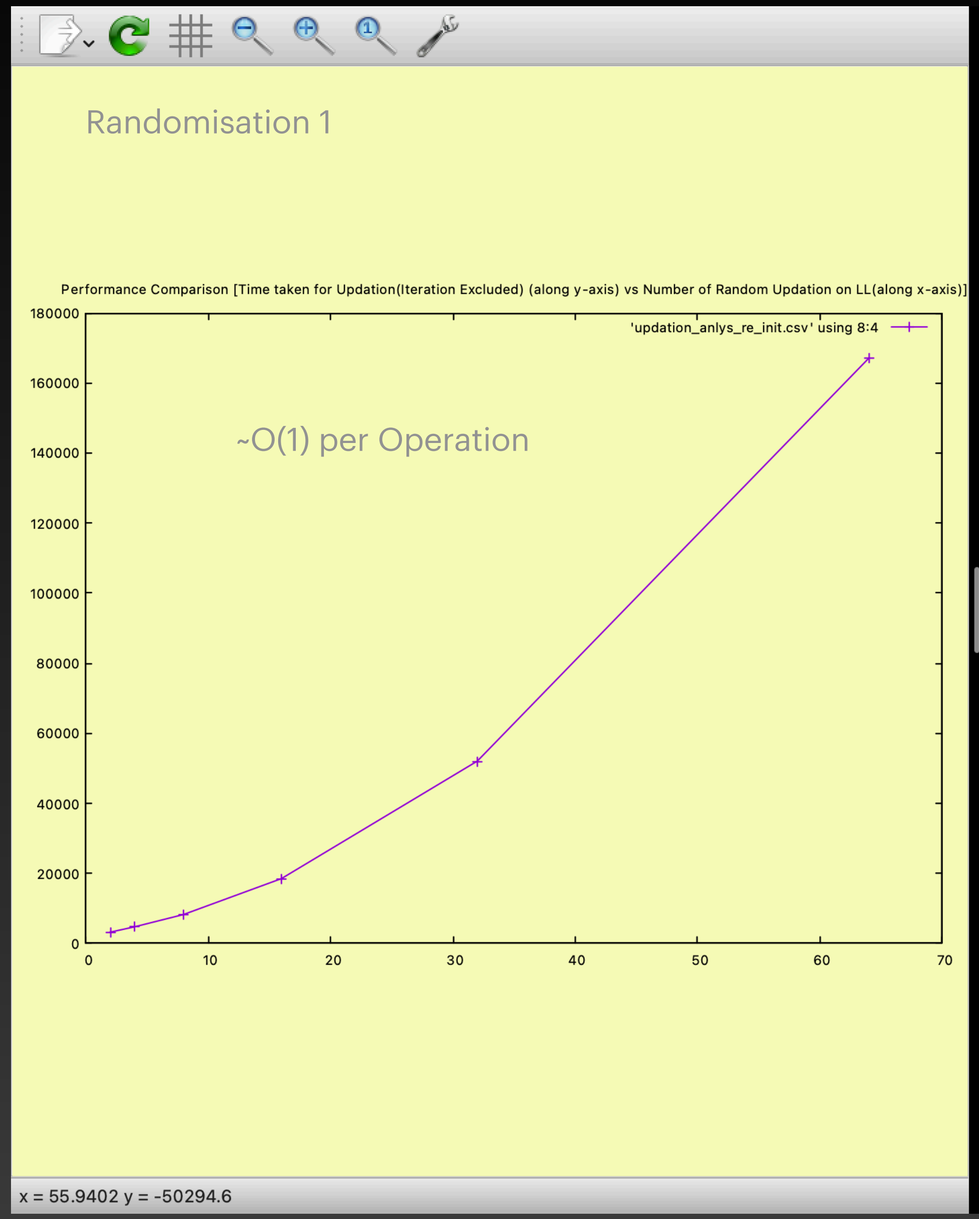
Strategies	Category 1	Category 2
Fat Node	Node Size: # ~ 20byte	O(N + V) to Hold The LinkedList
Path Copying	Node Size: # ~ 12 byte	Best Case: O(N) + O(V) Worst Case: O(N^2) + O(V) to Hold The Tree and Staring Pointers
Pointer Machine	Node Size: # ~ 140 byte	O(V) Amortised to Hold The LinkedList
Ephemeral Linked List	Node Size: # ~ 12 byte	O(N) to Hold The LinkedList

to store 4 byte Integer | 8 Bye pointers

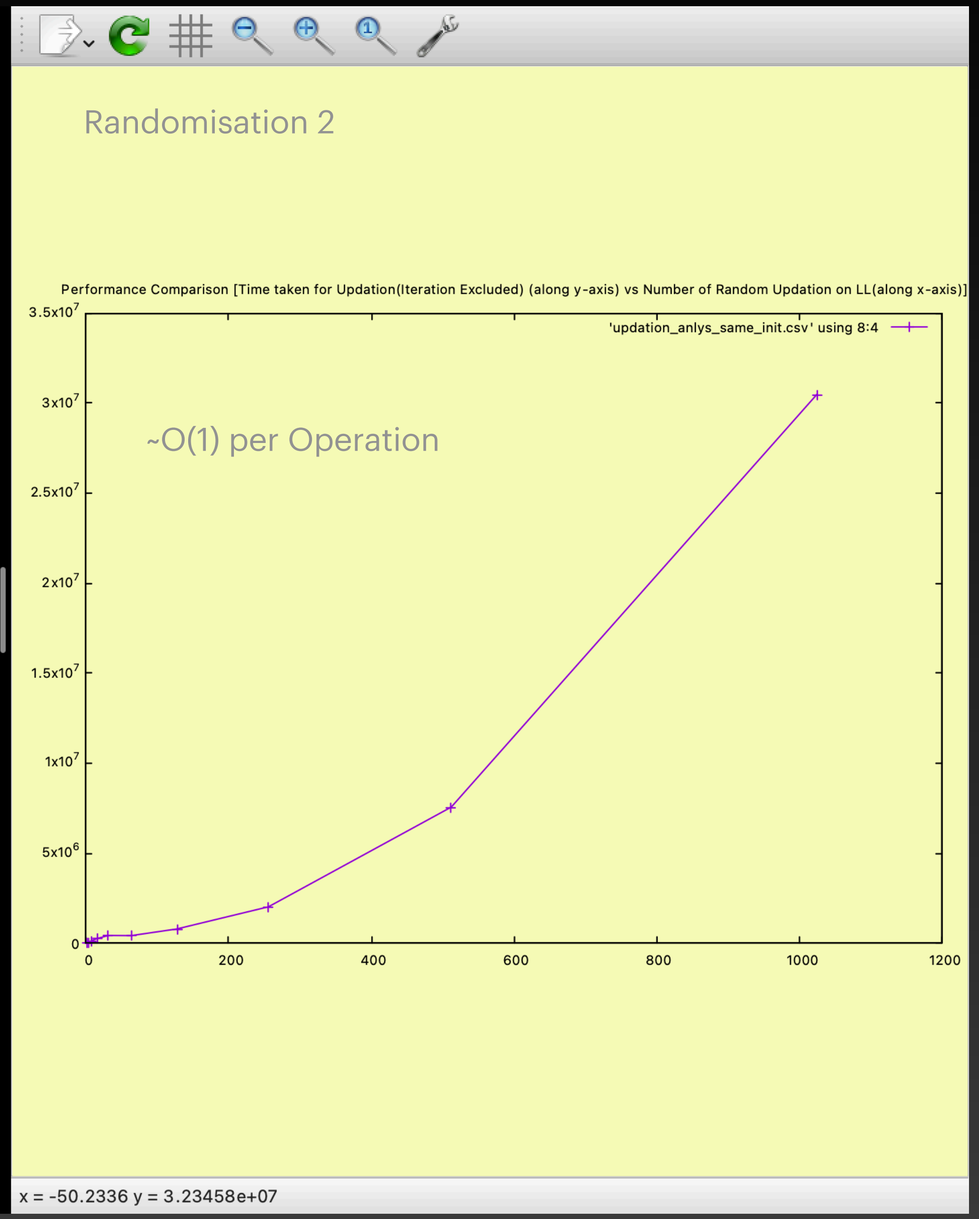
Benchmarking Of Pointer Machine Model

Updation (Value/ Insertion/ Deletion) Randomised

Randomisation 1

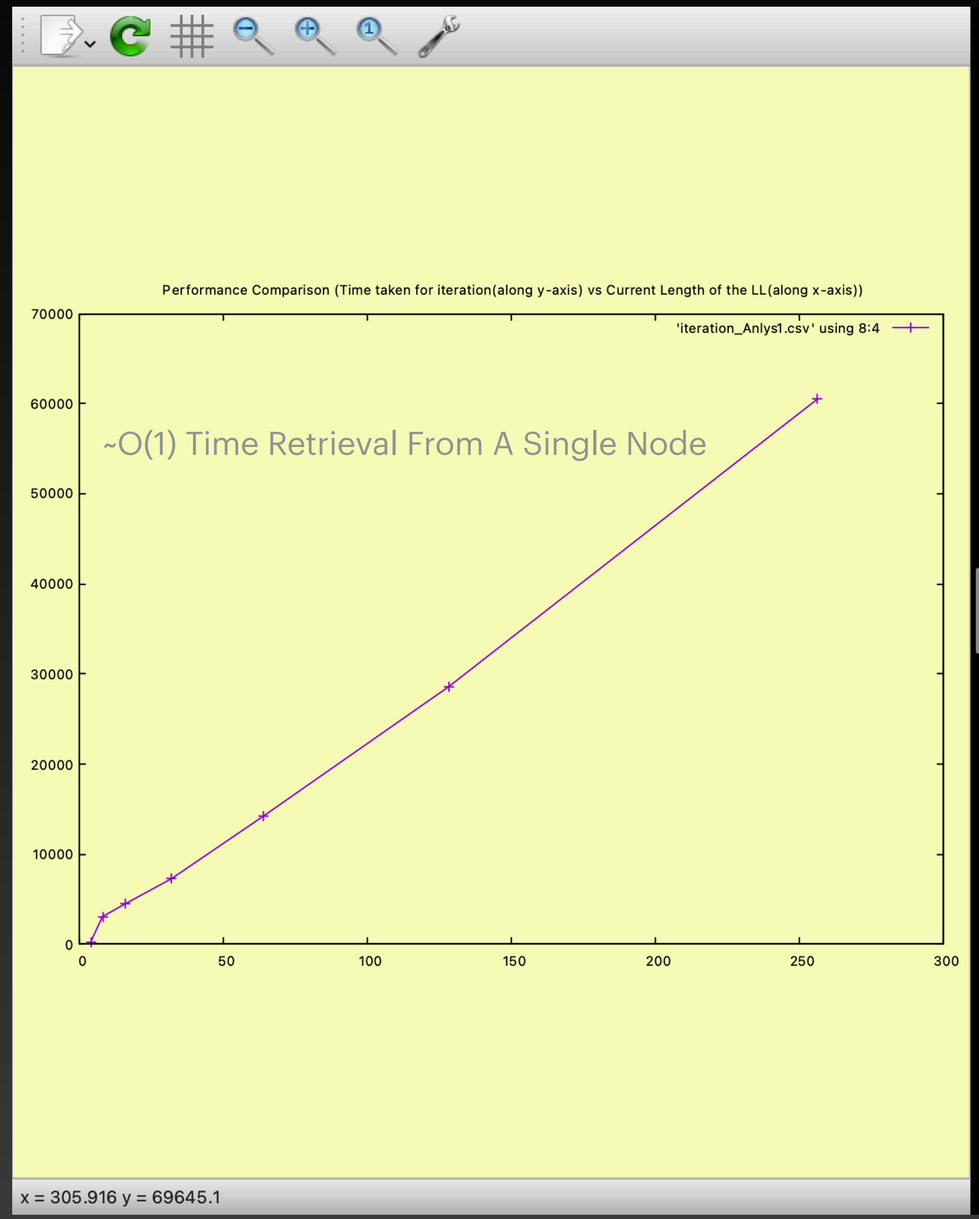


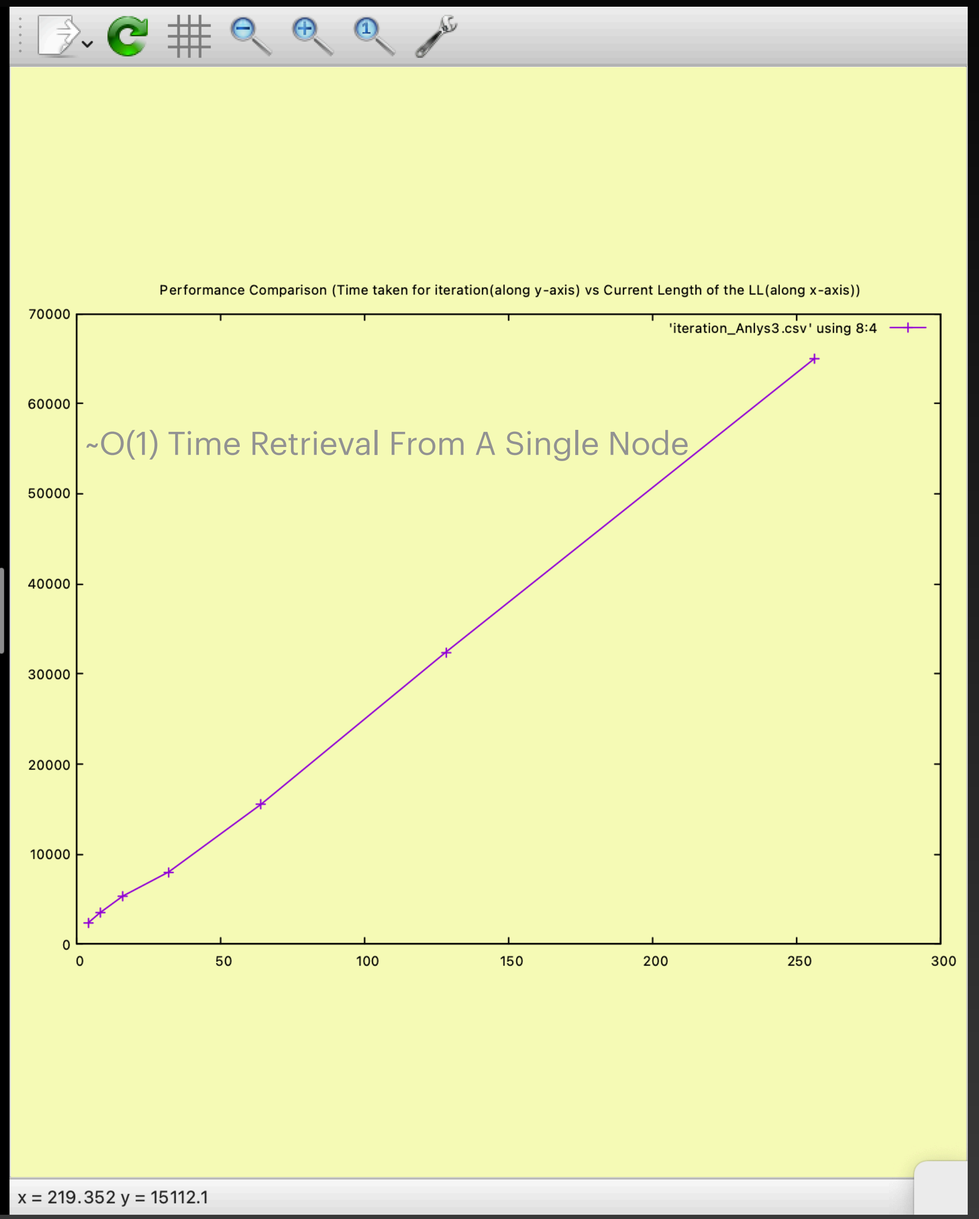
Randomisation 2



Benchmarking Of Pointer Machine Model

Iteration Through
Whole LL
At
Randomised Versions





Full Persistent Linked List

Full Persistent Linked List

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions

Strategies	InsertAfter(position,ver) #	DeleteAfter(position,ver) #	UpdateData (position,ver) #	RetrieveData (postition,version)	traverseWholeLL atVer(version)
Fat Node Pointer Machine [With List Maintenance]	O(1) [Amortized] + O(1) [Amortized] [amortised O(1) due for look up and insertions at version tree]	O(1) [Amortized] + O(1) [Amortized] [amortised O(1) due for look up and insertions at version tree]	O(1) [Amortized] + O(1) [Amortized] [amortised O(1) due for look up and insertions at version tree]	O(N) + O(1) [Amortized] [additional amortised O(1) due for look up at version tree]	O(N) * O(1) [additional amortised O(1) due for look up / insertions at version tree]
Path Copying	O(1) for at front O(N) for at Rear	O(1) for at front O(N) for at Rear	O(1) for at front O(N) for at Rear	O(N)	O(N)
Pointer Machine [With List Maintenance]	O(1) [Amortised]	O(1) [Amortised]	O(1) [Amortised]	O(N)	O(N)
Ephemeral Linked List	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(N) [Only Current Version Supported]	O(N) [Only Current Version Supported]

Full Persistent Linked List

Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Linked List Considering All The Versions

Strategies	Category 1	Category 2	Category 3
Fat Node Pointer Machine [With List Maintenance]	Node Size: # ~ 20byte	O(N + V) to Hold The LinkedList	O(V) to hold the Version Tree Here A ScapeGoat Tree
Path Copying	Node Size: # ~ 12 byte	Best Case: O(N) + O(V) Worst Case: O(N^2) + O(V) to Hold The Tree and Staring Pointers	-
Pointer Machine [With List Maintenance]	Node Size: # ~ 200 byte	O(V) [Amortised] to Hold The LinkedList	O(V) to hold the Version Tree Here A ScapeGoat Tree
Ephemeral Linked List	Node Size: # ~ 12 byte	O(N) to Hold The LinkedList	-

to store 4 byte Integer | 8 Bye pointers

Persistent Stack

Persistent Stack

Model Developed By Our Team

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Stack Considering All The Versions

Strategies	push(data, version)	pop(version)	getTop(ver)
Using PPL with PM Model	O(1)	O(1)	O(1)
DAG Model	O(1)	O(1)	O(1)
Ephemeral std::stack (C++)	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]

Persistent Stack

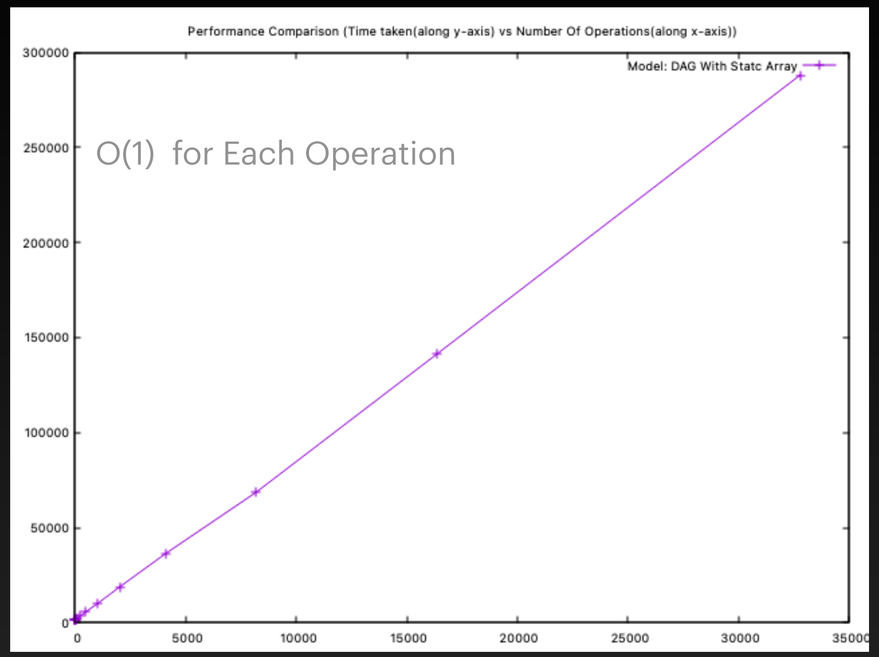
Model Developed By Our Team

Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Stack Considering All The Versions

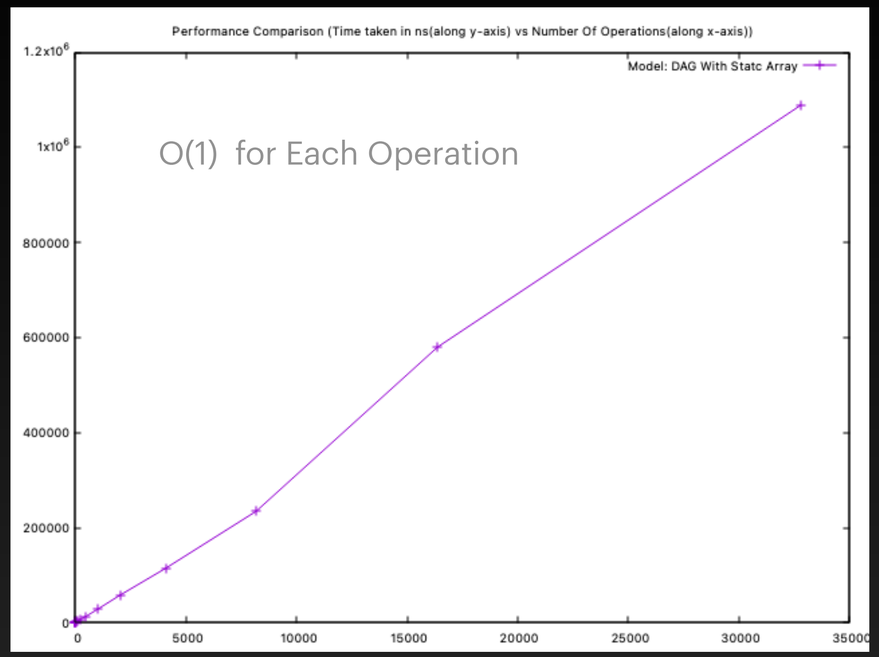
Strategies	Category 1	Category 2	Category 3
Using PPL with PM Model	O(V) [Amortized] To Hold The Linked List		
DAG Model	O(V) To Hold The MAP/DAG		
Ephemeral std::stack (C++)	O(N)	-	-

Benchmarking Of DAG Model

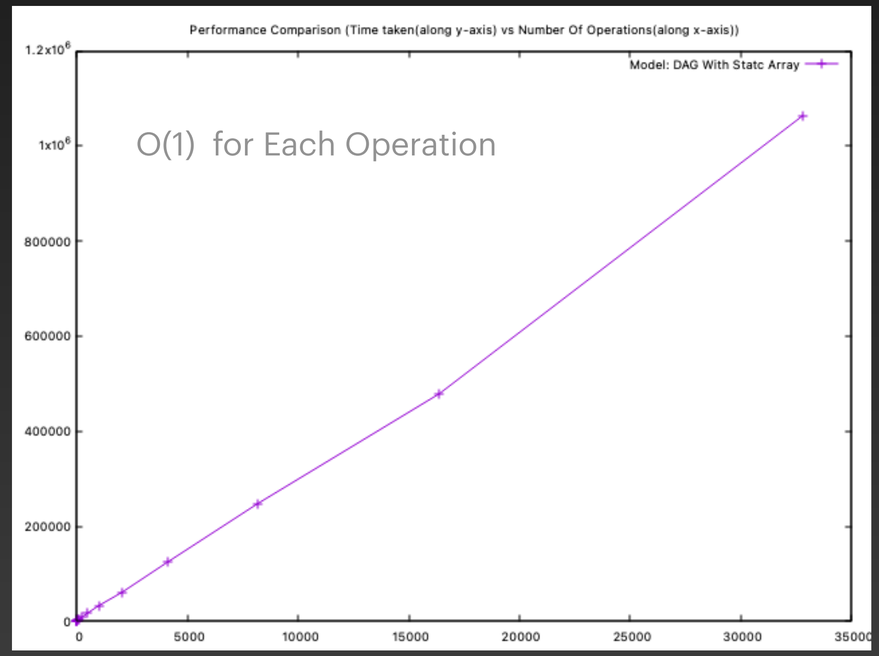


OnlyPush In Randomised Versions

Simulation Of Full Persistence

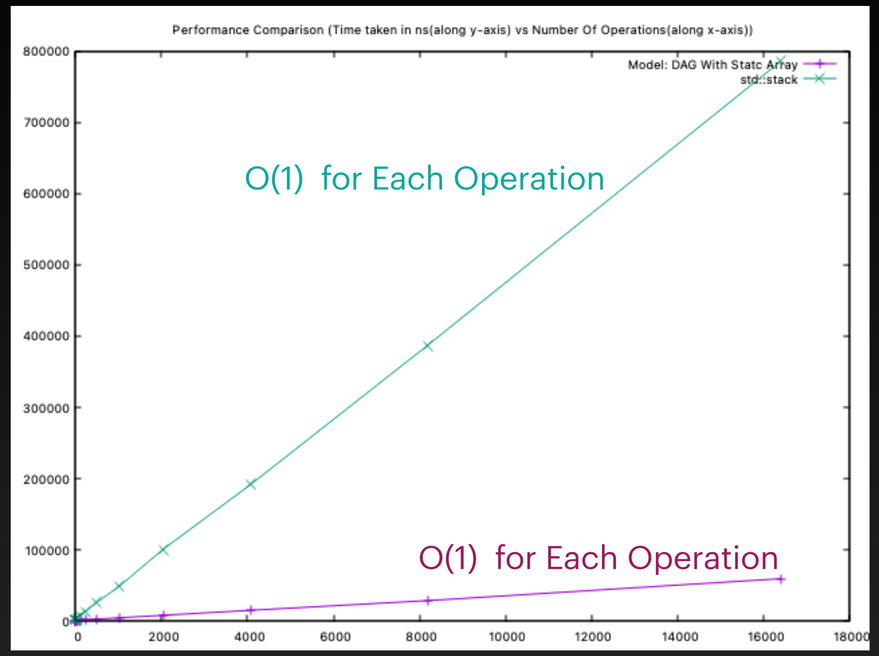


Randomised Push/Pop In Randomised Versions



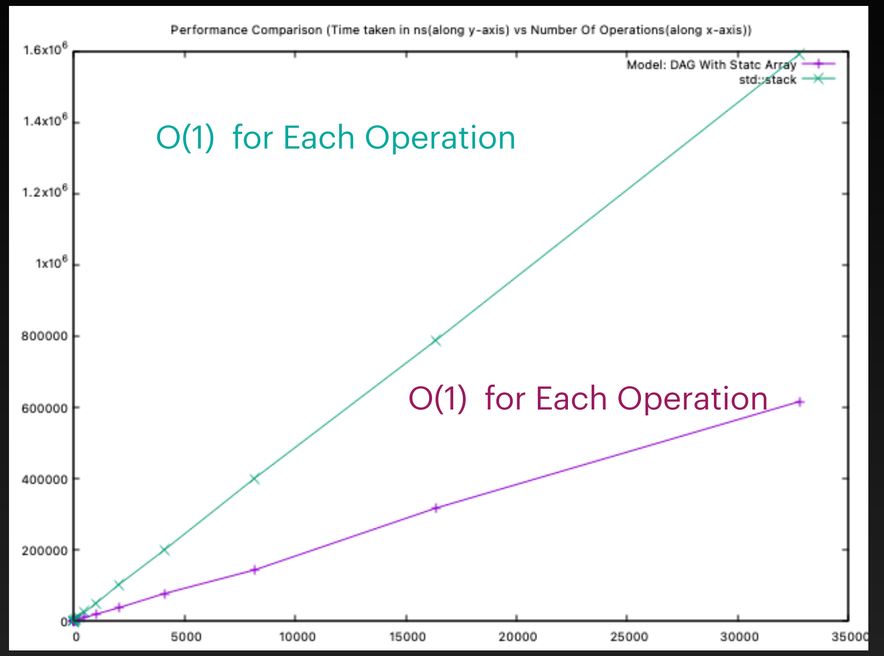
OnlyPop In Randomised Versions
(Pushing Operation is Not Time Profiled)

Benchmarking: DAG Model Vs std::stack

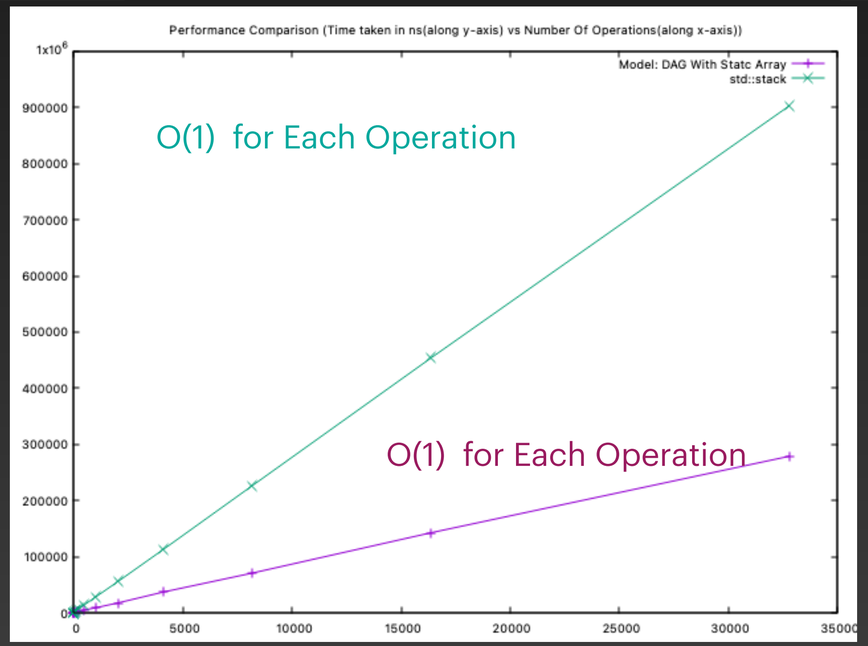


OnlyPush In latest Versions

Simulation Of Partial Persistence



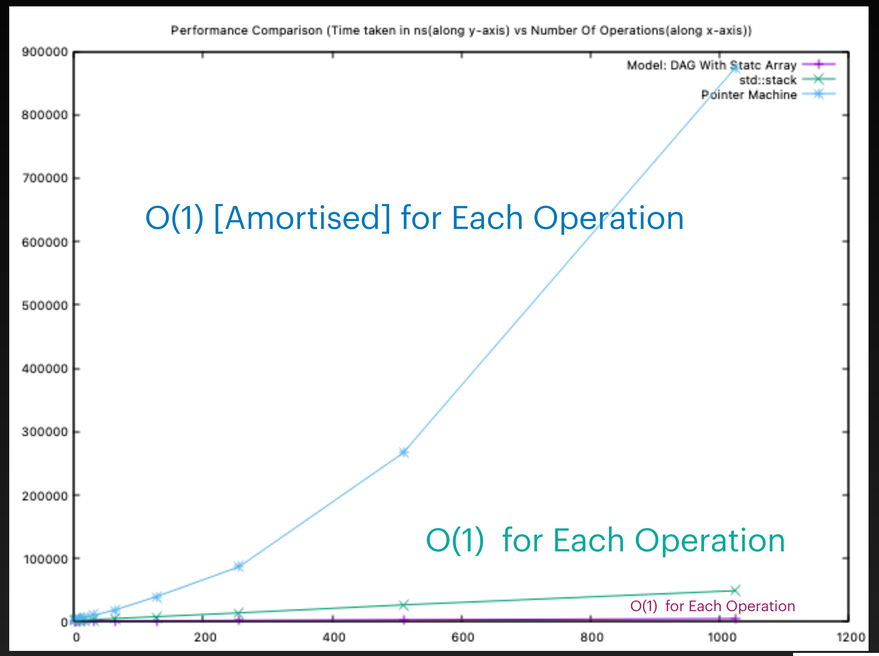
Randomised Push/Pop In Latest Versions



NOTE: We Used Static Array To Handle In DAG Model, So, It Is Slightly Faster Than Std::Stack

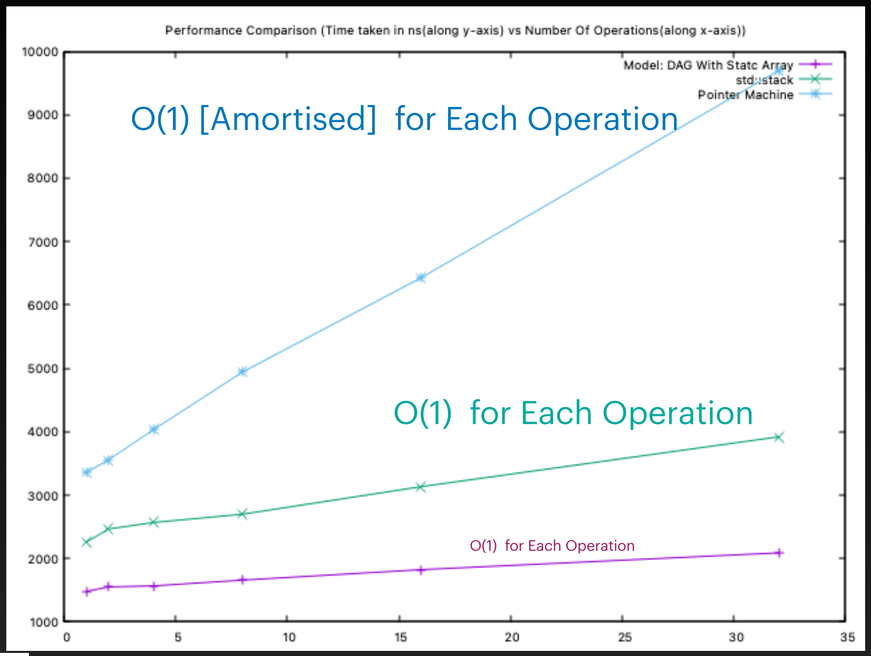
OnlyPop In Latest Versions
(Pushing Operation is Not Time Profiled)

Benchmarking: DAG Model Vs std::stack Vs Stack_Using_PM_PPL

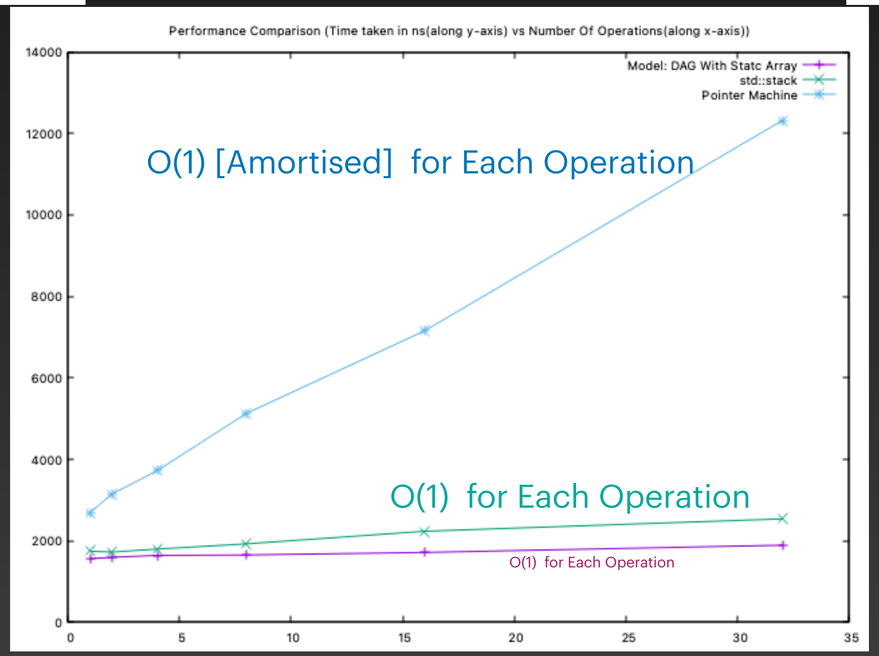


OnlyPush In Latest Versions

Simulation Of Partial Persistence



Randomised Push/Pop In Latest Versions



NOTE: We Used Static Array To Handle In DAG Model, So, It Is Slightly Faster Than Std::Stack

OnlyPop In Latest Versions
(Pushing Operation is Not Time Profiled)

Persistent Queue

Persistent Queue

Model Developed By Our Team

Time Complexity

Here, V = Total No. Of Version, N = Average Length Of The Queue Considering All The Versions

Strategies	enqueue(data, version)	dequeue(version)	getFront(ver)	getRear(rear)
Using PPL with PM Model	O(1)	O(1) [Amortized]	O(1)	O(1)
Reduced Sparse Matrix Model	O(log v) [Here v is current version] O((log V!)/V) ~ O(1) [Amortized]	O(log v) [In Average Case] O(v * log v) [In Worst Case]	O(1) [In Average Case] O(log v) [In Worst Case]	O(1)
Ephemeral std::queue (C++)	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]	O(1) [Only Current Version Supported]

Persistent Queue

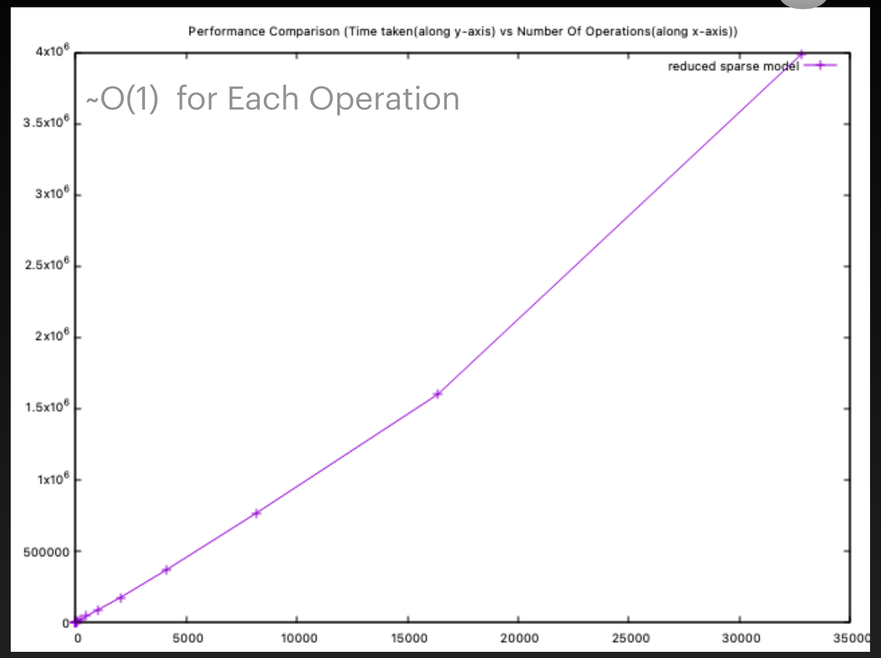
Model Developed By Our Team

Auxiliary Space

Here, V = Total No. Of Version, N = Average Length Of The Queue Considering All The Versions

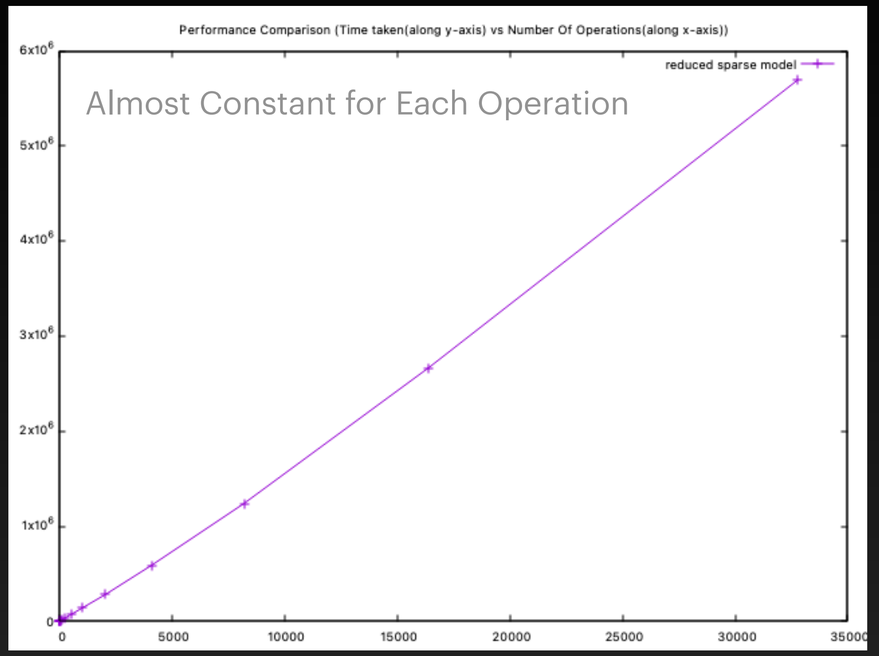
Strategies	Category 1	Category 2	Category 3
Using PPL with PM Model	O(V) [Amortized] To Hold The Linked List	O(V) to Hold The MAP for version->rear	
Reduced Sparse Matrix Model	• O (V + log (V!)) to Hold The UP_TABLE	• O (V) to Hold the MAP	• O (V) to Hold the TYPE_OF_VER
Ephemeral std::queue (C++)	O(N)		-

Benchmarking Of Reduced Sparse Matrix Model

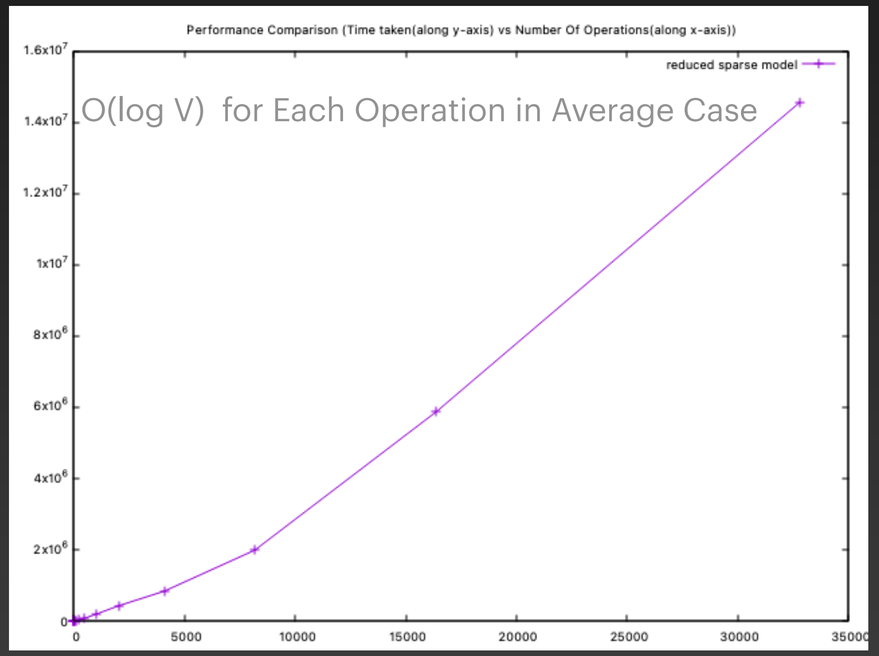


Only Enqueue In Randomised Versions

Simulation Of Full Persistence

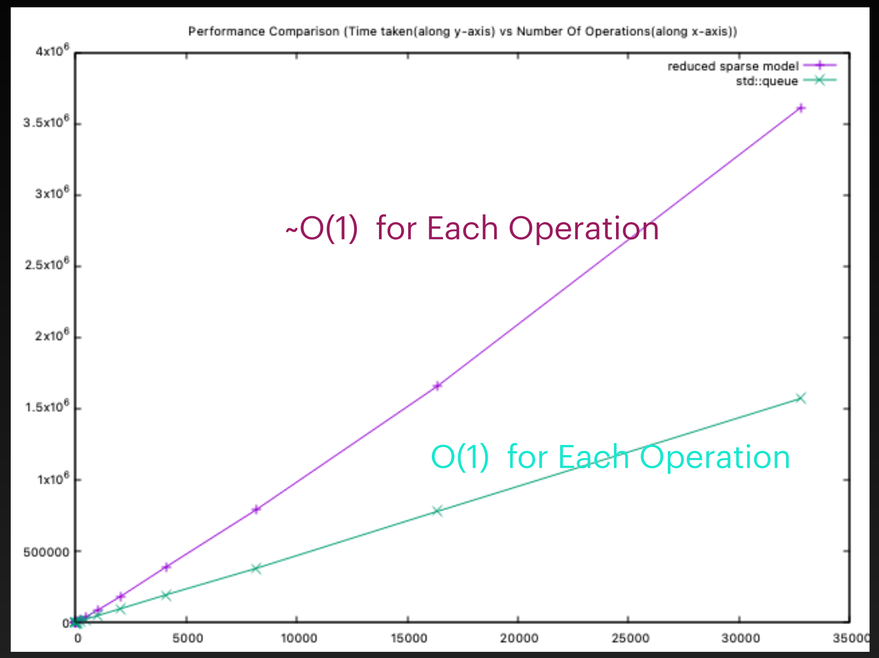


Randomised Enque/Deque In Randomised Versions



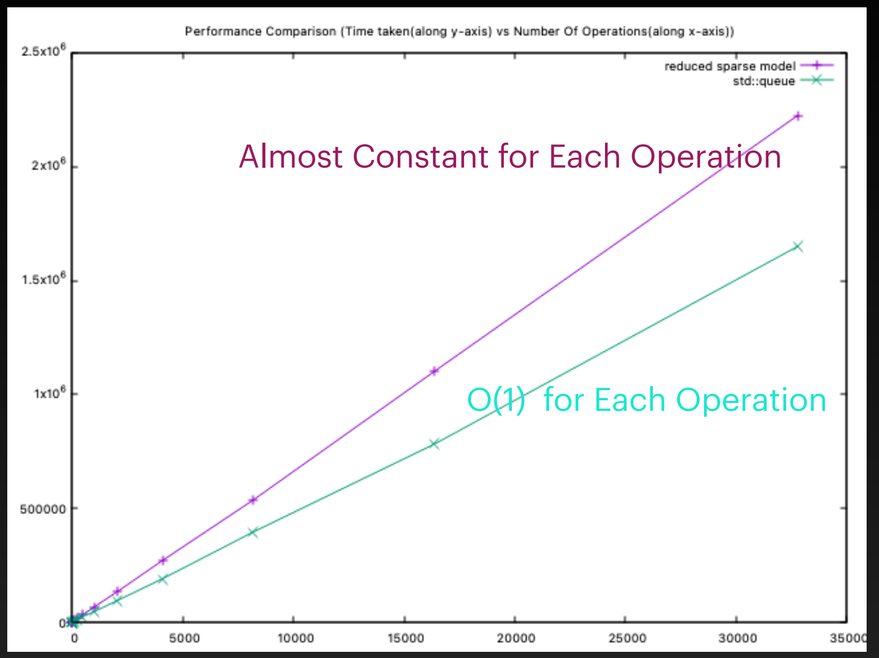
Only Deque In Randomised Versions
(Enqueuing Operation is Not Time Profiled)

Benchmarking: Reduced Sparse Matrix Model vs std::queue

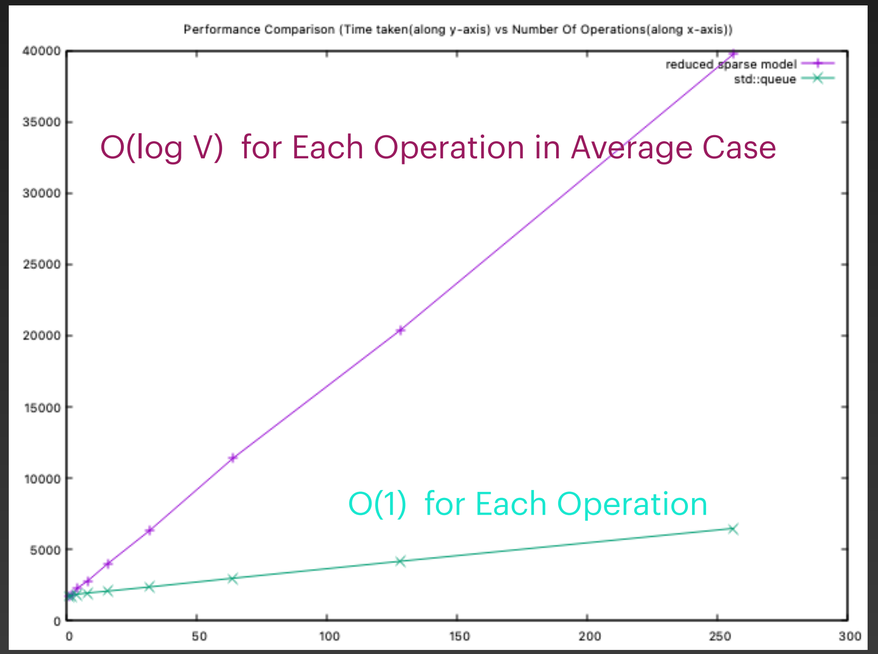


Only Enqueue In Latest Versions

Simulation Of Partial Persistence

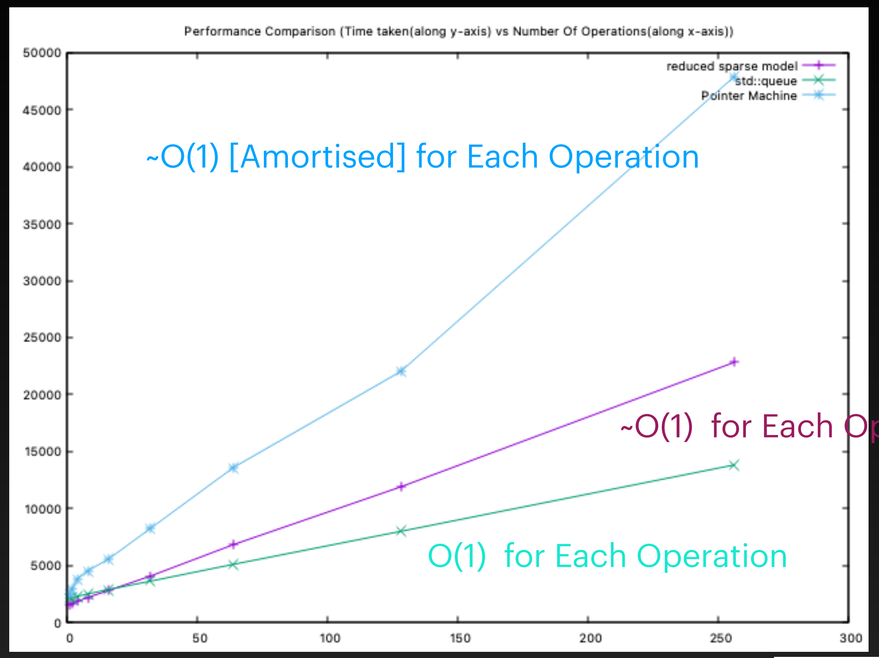


Randomised Enque/Deque In Latest Versions



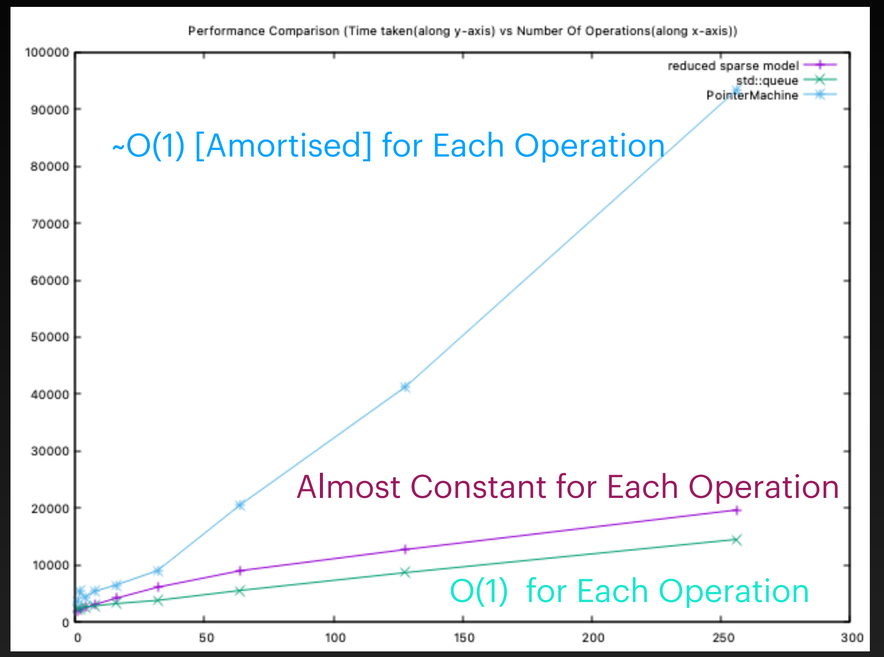
Only Deque In Latest Versions
(Enqueuing Operation is Not Time Profiled)

Benchmarking: Reduced Sparse Matrix Model vs std::queue Vs Queue_Using_PM_PPL

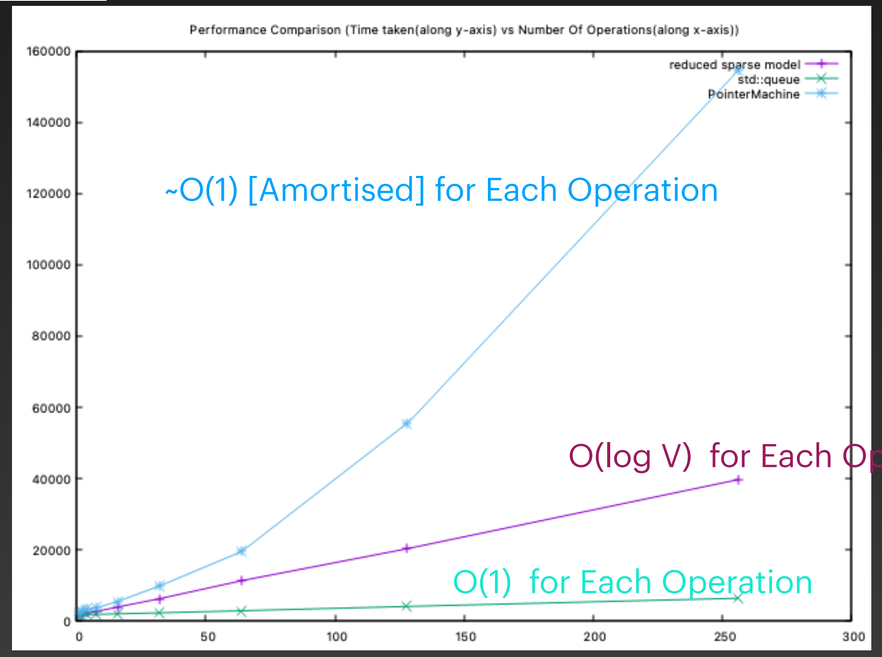


Only Enqueue In Latest Versions

Simulation Of Partial Persistence



Randomised Enque/Deque In Latest Versions



Only Deque In Latest Versions
(Enqueuing Operation is Not Time Profiled)

Persistent Search Tree

Persistent Search Tree

Time Complexity

Here, V = Total No. Of Version, N = Average Number Of Elements in The Tree Considering All The Versions

Strategies	Updation/Insertion/ Deletion of Data	Retrieval Of Data
Path Copying With Normal BST	O(N) in Worst Case O(1) in Best Case	O(V) + O(N) in Worst Case O(V) + O(1) in Best Case
Path Copying With AVL/RB Tree	O(log2N) in Worst Case O(1) in Best Case	O(V) + O(log2N) in Worst Case O(V) + O(1) in Best Case
Path Copying With Hash Array Mapped Trie	O(log32_(2^64))~O(12)	O(V) + O(log32_(2^64))~O(12)
With Pointer Machine	O(1) Amortised	O(1) Amortised

Persistent Search Tree

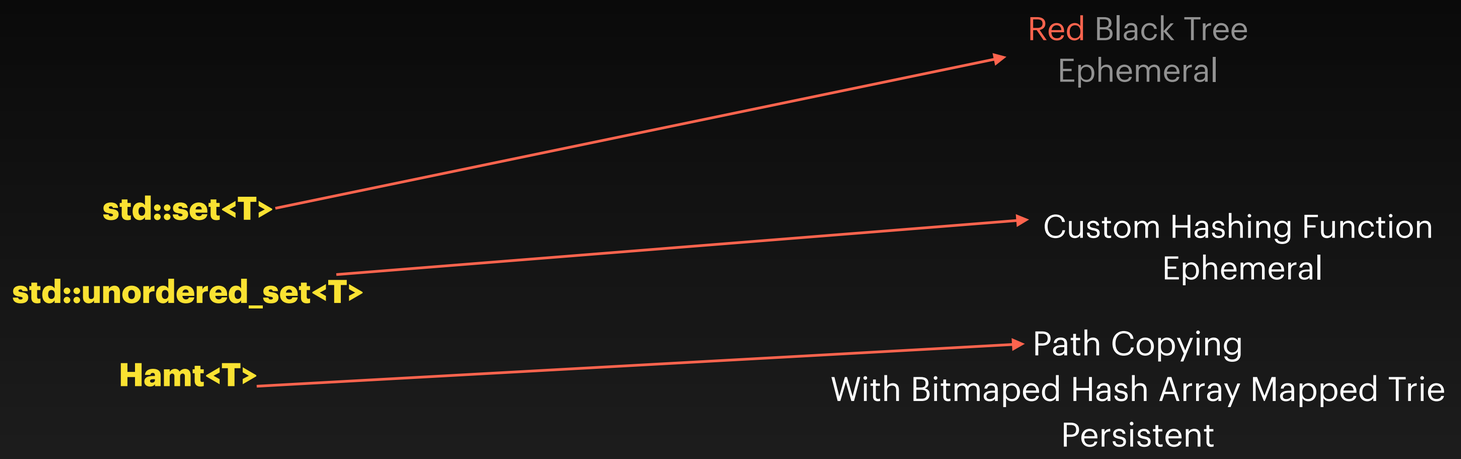
Auxiliary Space

Here, V = Total No. Of Version, N = Average Number Of Elements in The Tree Considering All The Versions

Strategies	Category 1	Category 2
Path Copying With Normal Threaded BST	Node Size: # ~ 28 byte	O(N + V) to Hold The Tree and Staring Pointers
Path Copying With AVL/RB Tree	Node Size: # ~ 32 byte	O(N + V) to Hold The Tree and Staring Pointers
Path Copying With Bitmaped Hash Array Mapped Trie	Node Size: # ~ (4+4+64*8) byte [Worst Case] ~ (4+4) byte [Best Case]	O(N + V) to Hold The Tree and Staring Pointers
With Pointer Machine	Node Size: # ~ 250 byte	O(N) [Amortised] to Hold The LinkedList

to store 4 byte Integer | 8 Bye pointers

BenchMarking Of Search Tree



BenchMarking Tool:

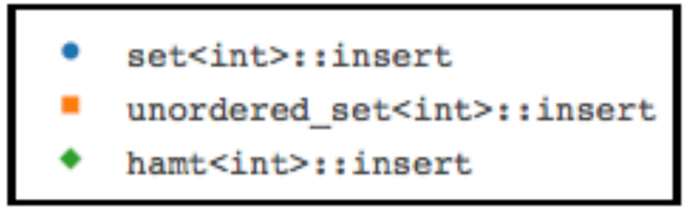
Nonius

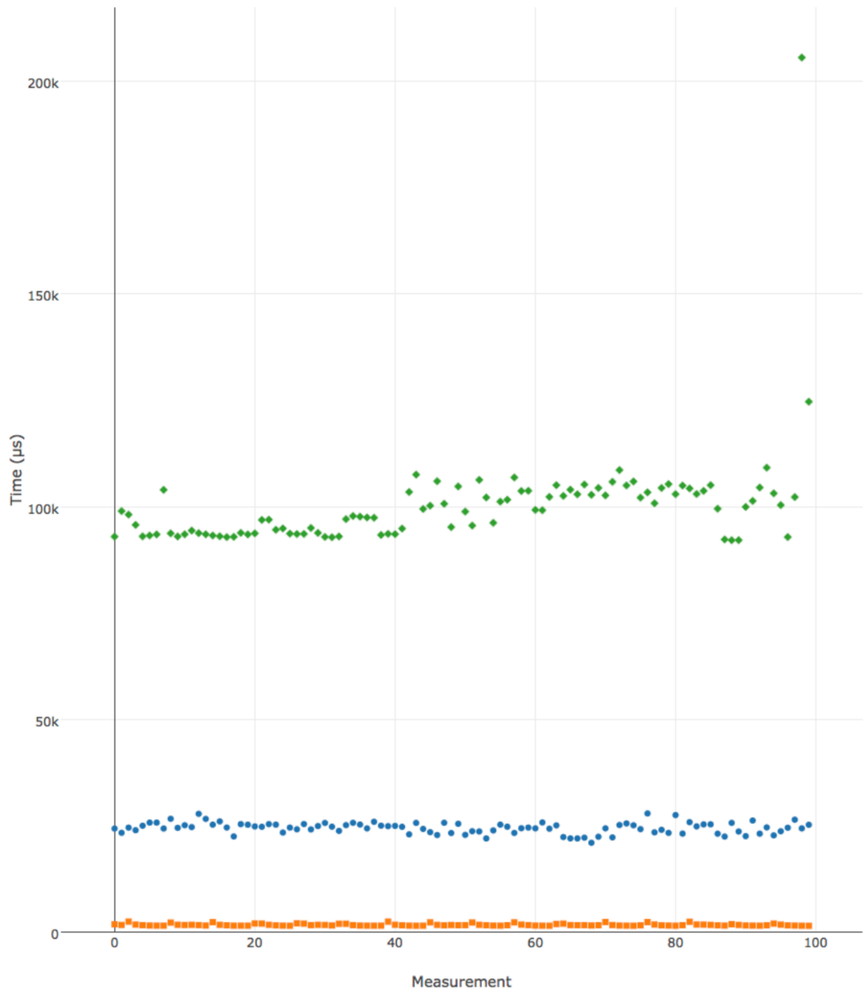
A C++ micro-benchmarking framework

Benchmarking Data Taken From A Talk By
Phil Nash,
Developer Advocate

BenchMarking Of Search Tree

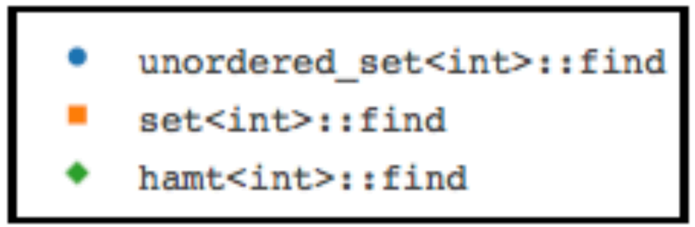
insert
100k ints

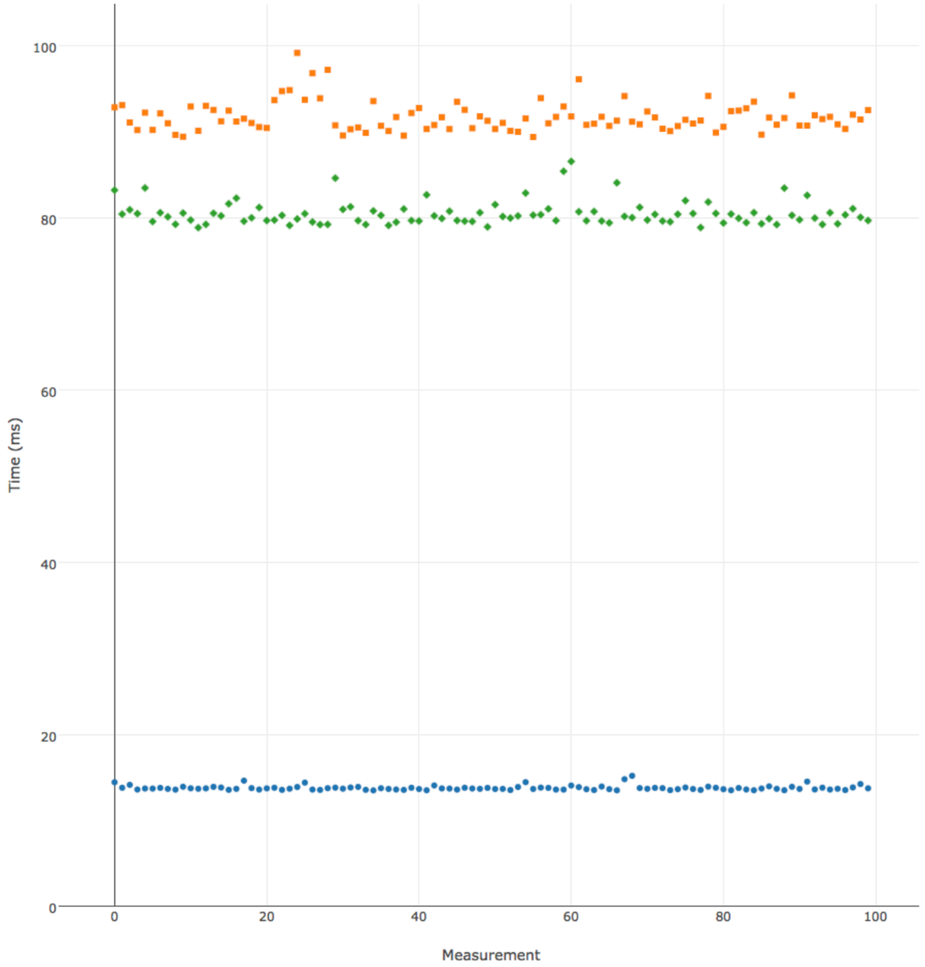




BenchMarking Of Search Tree

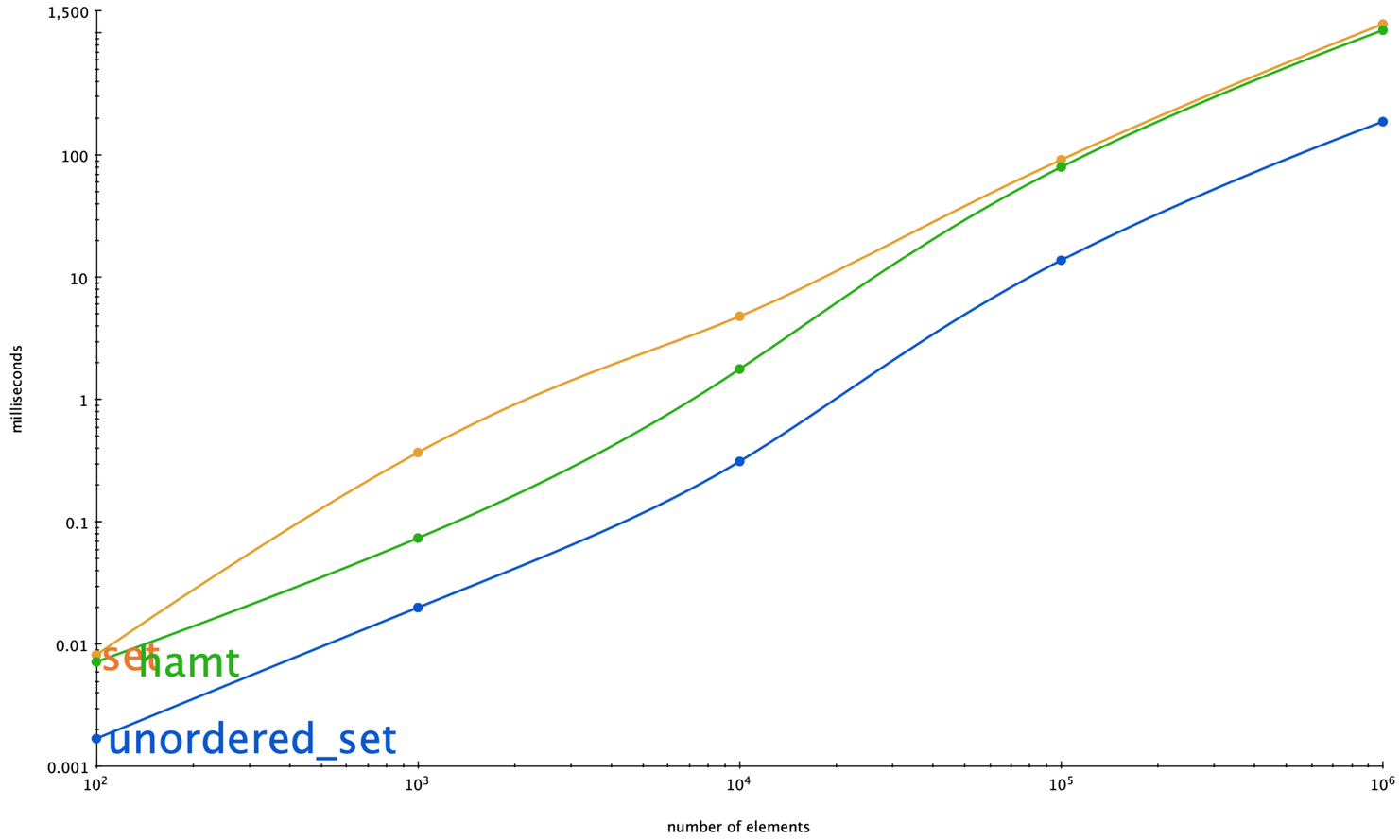
find
100k ints





BenchMarking Of Search Tree

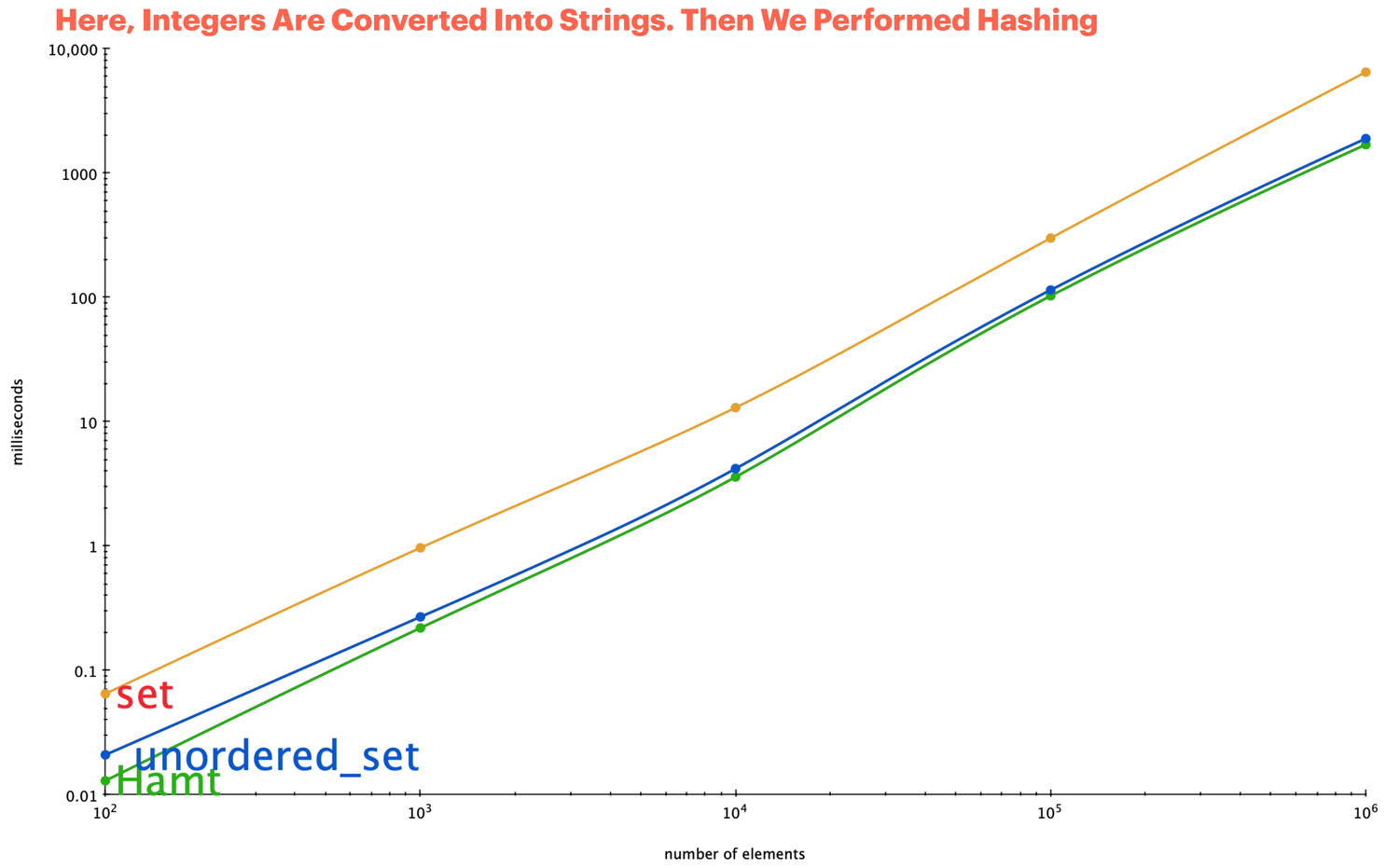
find
100-1m
ints



OmniGraphSketcher

BenchMarking Of Search Tree

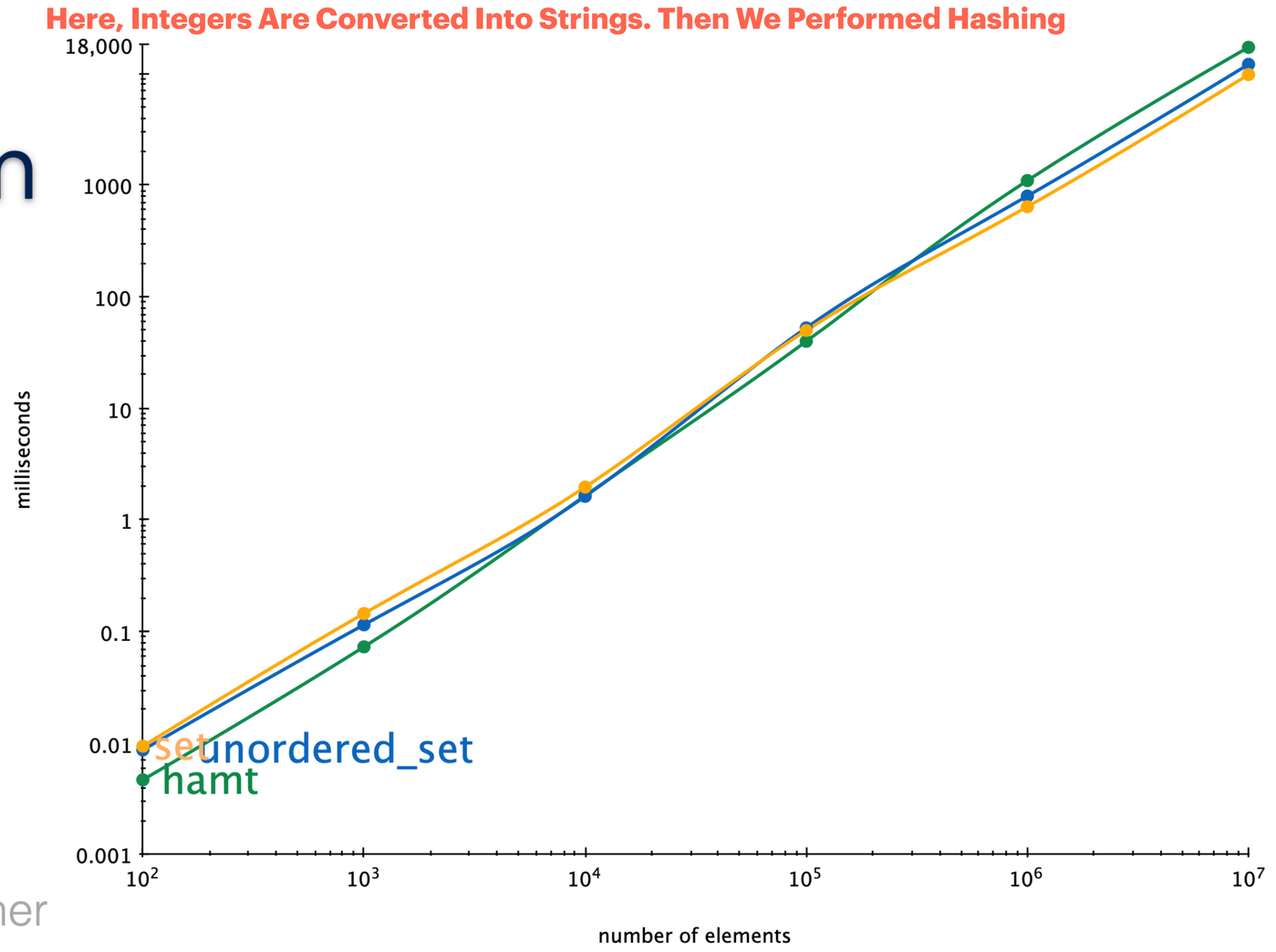
find
100-1m
strings



OmniGraphSketcher

BenchMarking Of Search Tree

find
100-10m
hashes



OmniGraphSketcher