Comparative Analysis of Different Binary Tree and Priority Queue (Heap) Algorithms

Fakhruddin Amjherawala
Research Scholar
Sage University, Indore
Assistant Professor
IPS Academy, Indore, India

Sanjay Dubey, PhD
Head of Institute
Associate Professor
Institute of Computer Application
Sage University, Indore, India

Ummulbanin Amjherawala
Assistant Professor
School of Computers
IPS Academy, Indore, India

ABSTRACT
A tree is the core building block to arrange data in a specific order. Different tree structure arrangement gives the capability to store, retrieve, rearrange, find, and free the data more efficiently. Numerous algorithms build to satisfy the overall arrangement of tree data structure to minimize the complexity in terms of time and space. The priority queue Algorithm uses the tree structure to give the arrangement a direction so that data must sort and place according to its priority. Sequencing in priority impacts the mechanism to store and retrieve the data. In this paper comparative study perform on different tree data structures and how it will be beneficial when the tree structure merges with the priority queue.

General Terms
Algorithms, Tree.

Keywords
Binary Tree, Binomial, Fibonacci, Index, Heap, Priority.

1. INTRODUCTION
In today's world data and its organization play a crucial role in the field of IT. In every field managing, filtering, and predicting forthcoming events based on previously well-organized structured data as well as there is a challenge to form data arrangement efficiently, there are many algorithms implemented to improve the efficiency of storing and retrieving data as fast as possible.

Analyses the cache performance of the priority queues, As the difference in execution time between a cache miss and hit can be anywhere from twofold to tenfold or more[1], rather than viewing priority queues as just the sum of their operations, This will be done by considering the priority queue implementations in terms of the Principles of Locality.

1.1 Principles of Locality
The principles of locality are based on observations regarding memory access patterns. These observations can be summarized as follows: memory access patterns are not random, accesses tend to repeat themselves in time, and accesses are often followed by accesses to nearby addresses in the memory address space [2]. The principles of locality are split up according to the two observations Spatial Locality and Temporal Locality.

1.1.1 Spatial Locality
Spatial locality refers to the phenomenon that memory accesses are often followed by accesses to addresses nearby in memory space. This arises from the programmer's and the compilers' tendency to lump objects together [1 2]. A good practical example of this is arrays. Arrays are allocated as contiguous blocks in memory, if access data at the index i, likely going to access the data at the index i+1 shortly. An obvious way to optimize this so that not to pull every single piece of data into the cache just as need it is to use some sort of look ahead. Processors implement this via something called cache lines. Cache lines are atomic storage units in the cache which are larger than most primitive data types, in modern processors cache lines are typically 64 Bytes. When data is accessed and loaded into the cache, nearby data is also loaded into the cache to fill up the entire cache line.

1.1.2 Temporal Locality
Temporal locality refers to the phenomenon that memory accesses tend to repeat themselves in time. It is strongly intertwined with spatial locality in the sense that the prediction here is not only that nearby data will be accessed in the future but the same piece of data. Support for this is implemented by the very existence of caches, or the fact that data is loaded from main memory into the cache. Cache replacement policies such as LRU also adhere to this by evicting the least recently used cache line in the cache of a write miss.

In this study, compare and analyze different Binary tree implementations along with the role of it in the implementation of Priority Queue with execution time.

2. LITERATURE REVIEW
2.1 Binary Tree
A binary tree [1] is a core building block data structure. In a binary tree element arrangement and its representation can be easily understood. The element defines in the form of node and its organization reflects either a full or complete binary tree. Visit element in the form of root to a child or vice versa.

While implementing it requires lots of storage space like using Arrays, recursion, lots of movement required while setting and removing elements, and visiting the child repeating root node twice, due that its execution time going high.

2.2 Binary Search Tree
In the Binary Search tree [1] with the feature of Binary tree focus on data ordering, so that less than or greater than property merged in the sub-node (child), left sub node follow less than and right sub node follows greater than the rule, each sub-tree recursively follow this property while storing and searching the data.

While implementing it requires lots of storage space like using recursion (stack space), lots of movement is still there while fetching the entire element. There is another issue related to its structure depending on what element are arranging, so there

may be the chances of either a left-skewed binary search tree or may be right skewed binary tree structure built due to less than or greater than a rule, which makes their arrangement linearly.

2.3 AVL Tree

In AVL tree [3] proposed an approach to a nearly balanced binary search tree, an optimal shape achieved by associating rebalancing factor, the tree loses its shape while randomly inserting and deleting node elements. Maintain its height by checking the balancing factor of the left and right sub-tree not more than 1 [4].

Inserting data into the binary tree is just a Binary search tree, but at the time new node insertion checked the balance, if it breaks the rule then applies the single and double rotation to rebalance the tree. According to it, if the left sub-tree is left-skewed, then put up the single right rotation, and if it is right skewed then put up the left rotation.

Efficiency in terms of time needs $O(\log n)$ [5] when each time addition or deletion is performed due to checking of balance factor to balance the tree. On the other side, efficiency in terms of space to store the balance factor of each node. The height of the tree is at most $1.44(\log n)$, so time to search approx. $\log n$ and when the size of n is an average number of comparisons for searching about $\log n = 0.25$ [6].

2.4 Red-Black Tree

In the Red-Black tree [7 8] nodes are either red or black, the root is always black, if there is a red node then its child never be red, while inserting a new node of red color and its parent node is black otherwise change the parent node color and sibling to black and its immediate parent node to red.

Compare with the AVL tree it requires less comparison during deletion operation but the height is increased. Assume n number of nodes in a red-black tree. The minimum and maximum height would be $\log_2 n + 1$and $2 \log_2 n + 1$. If the height is h then the minimum and maximum number of nodes are $2^{h/2}$-1 and 2^h-1. On the other side, when n nodes in an optimal balanced binary search tree then the height would be $\log_2 n + 1$. The number of nodes in it of height h is between 2^{h-1} and $2^h - 1$[9].

2.5 Optimizing BST by rebalancing with Sorting Algorithm

In optimizing the Binary Search tree [7] try to improve the result as compared with the AVL tree and Red-Black tree.

In Binary search tree insertion create $n!$ Number of different structures (there is n number of nodes in a tree) with the same data but different ordering. In this method apply basic Binary search tree insertion along with data arrangement (sorting) to balance the tree.

In AVL and other methods rearrange the tree after each operation, in this method get the median by traversing the tree using stack. The time complexity is $O(n)$ because of rearranging each node at least once to achieve the optimal shape. Memory Space is also one of the factors by the use of the stack to show optimal tree structure. Still, there is an issue of unbalance, so there is a need to rebalance it [7].

2.6 Threaded Binary Tree

Threaded binary makes use of null value to improve the traversal process and tries to utilize the space of null values (wasted) in the Binary Search tree because it points to nothing. They add the concept of an in-order predecessor to utilize the node left null pointer, in the same way as the node right pointer which points to an in-order successor [8]. It means null are replaced by references of other nodes which are called threads.

To visit a node, in the threaded binary tree there are at most three paths, but in the binary search tree, there is one path. If attaching a thread, the Boolean value there set to true otherwise false, for the differentiation of either actual link or threaded link. This Boolean value breaks the infinite processing. The left pointer of the leftmost node and the right pointer of the rightmost node are null.

There are three ways to implement a thread:

2.6.1 One-way threading (Single Threaded)

Where a right child is NULL, it refers to its in-order successor.

2.6.2 Two-way threading (Double Threaded)

Where both a left child and a right child are NULL, it refers to its in-order predecessor and in-order successor respectively.

2.6.3 Two-way threading with the header

Those are either left-most or right-most nodes that do not have their predecessor or successor respectively, then their Null field refers to the header node which contains three fields, the left contains root node reference, the data field is blank, and the right refers to itself.

2.7 Balance Binary Search Tree Globally

In this approach rearrange the reference to balance the tree [10]. In this algorithm to rebalance, first, get the sorted data by traversing the tree using in-order recursively which is executed in linear time. The second approach is partitioning by the folding method [10].

This algorithm [10] first finds the order of nodes by traversing and storing the order's content into the list in such a way that i^{th} position compare with $i^{th} + 1$ position for less than and with i^{th}-1 position for greater than. Next, find the left median $\lfloor n\text{-}1/2 \rfloor$ and start constructing a tree with the root node. The values less than the left median are a part of the left sub-tree otherwise be a part of the right sub-tree. Left sub-tree and right sub-tree are continuously selected in the existing tree. This algorithm requires extra space as a List using an array which requires continuous memory which creates an issue when there is a lack of consecutive space in memory. To sort the data traverse each node which takes $O(n)$ time.

2.8 Splay Tree

The splay tree [11] is based on the principle of temporal locality; according to it 90% of the accesses are to 10% of the data items [12]. In splaying, frequently retrieve elements are adjusted and bring and set at the top of the tree for quick access [11].

In splaying several operations say, zig, zig-zig, and zig-zag are performed. To make it efficient there is no requirement to store the balance factor as additional data. Performance is high in the application where recently accessed data is required.

Accessing (running time) to a node becomes quite expensive in the case where the splay tree is highly unbalanced. While looking toward total running time into consideration then it is inexpensive (efficient) due to amortized performance $O(\log n)$. Worse time [6] complexity may be $O(n)$. The disadvantage is that more adjustment and within a sequence individual operation is there.

2.9 Day-Stout-Warren Algorithm

In FORTRAN recursion not there so Day's algorithm [10] introduced a threaded binary search tree, which was modified by tree rebalancing in optimal time and space [13].

In this approach, first Nodes are to be sorted, to achieve this prepare a skeleton by performing right rotation until the left reference become null. Next, prepare the complete or almost complete binary tree by balancing from the top of the build skeleton.

In a complete binary tree, all nodes are exactly two children and all leaves are at the same level. Threaded binary trees require extra memory compared with Binary search trees. This algorithm first stores the initial tree, but compared with other techniques do not require much extra space.

Execution times of the number of nodes are linear. This algorithm does not require Stack and arrays to convert into an intermediate form from the tree. The overall running time of the algorithm is $O(n)$.

2.10 Anderson Algorithm

This is a red-black tree-based algorithm. It [14] includes the extra feature of the left child may not be red along with the existing red-black tree concept. It extracts half of the restructuring computation. Different rotations are required along with its deletion operation are trickery.

To ignore its colouring, each node level is stored and maintained during implementation. AA algorithm is simplified using skew and split operation. Remove left horizontal links called skew operation and remove consecutive horizontal links called split operation, balancing require 3 skews and 2 split operations. When deletion is more often required to simplify the operations with the rule of left child may not be red [9].

2.11 Heaps (Priority Queues)

In the priority queue, continuous additions and deletions operation try to find the greatest and lowest element and set it frequently on a priority basis.

One way to initialize arrays of elements, do a sequential search $O(n)$ time to find the smallest element and store it temporarily, any manipulation regarding the smallest element repeats the sequential search. Another way to sort the list of elements using merge sort in $O(n\log n)$ time. Any manipulation in priority and then sorting will take $O(\log n)$ time using binary search, but finding the highest priority element will be done in $O(1)$ time.

This however seems to be quite repetitive and expensive. If considered it is only the element with the lowest priority does the entire list sorted at all times? This issue can be resolved using heap more efficiently.

2.12 Implicit Heap

Implicit heaps are defined in arrays; this means that they are allocated as a contiguous block of memory which is great for the spatial locality. Explicit heaps on the other hand exhibit poor spatial locality as each node will be allocated separately.

When considering explicit heaps and caches, the higher constant factor is involved so that the explicit heaps will fill the cache quicker which results in more cache misses for equally sized heaps.

2.13 Binary Heap

A Binary Heap [15] is a form of a binary tree. The Binary Heap must satisfy an additional constraint, the shape property. The shape property implies that the Binary Heap must be a complete binary tree, that is, all levels of the heap must be filled, except for the last level, which is filled from left to right.

The Binary Heap has some nice properties due to the shape property. Every layer of the heap, except the root, has a layer with twice as many nodes above it. From this, it can be conclude that a node at index i, will have a parent with index $\lfloor(i-1)/2\rfloor$. Vice versa is true for the children of a node, they can be found at the index $i*2 + C$ where $C \in \{1,2\}$. This relationship between parent and child nodes allows to implicitly define the heap. Store all of the nodes in a contiguous array in the order that they are traversed through the tree from left to right, rather than explicitly defining the tree structure. This significantly reduces the memory overhead and improves the locality as mentioned in section 2.2

2.14 d-ary Heap

Generalized forms of binary heaps are d-ary heaps [16]. It contains d child nodes instead of 2 as with binary heap ($d = 2$). In the d-ary heap $d > 2$ which makes finding the upward node faster as it takes $\log dn$, the height of the tree, compare with the binary heap [16]. According to the selection of d, Cache performance is better in the d-ary heap, it runs faster in practice compared with the theoretical aspect of RAM model [17]. The best choice when d=4, is because of the improvement in the cache performance of the heap.

The implementation of a d-ary Heap is identical to that of a Binary Heap, it is just a generalization. A node at index i will have a parent with index $\lfloor(i-1)/d\rfloor$ and potential children will be at index $i*d + C$ where $C \in \{1,2,\dots,d\}$.

2.15 RS Tree

A randomized search Tree was given by Seidel et al. [18]. In this algorithm, nodes contain key values as a priority which chooses randomly. It maintains a binary search tree with priority.

An RS Tree may be max-heap or min-heap:

Max-heap: The parent node contains a larger value compared with its child node.

Min-heap: The parent node contains a smaller value compared with its child node.

For each insertion follow this step and rule [18]:

Step: As a Binary search tree insert data, here insert data with priority then rotation is applied to set (compare parent and child) node according to priority.

Rule:

a. If p is the left successor of q, then the value of node p will be less than the value in node q.

b. If p is the right successor of q, then the value of node p will be greater than the value in node q.

c. If p is a child of q, then the Priority assigned to node p will be greater or equal to the priority of q.

2.16 Binomial Tree

To understand Binomial Heap one must first understand Binomial Trees. A Binomial Tree of degree 0 is just a single node. A Binomial Tree of degree 1 is formed by connecting the roots of two Binomial Trees of degree 0. This process is recursive; a Binomial Tree of degree is formed by joining two Binomial Trees of degreek-1.

2.17 Binomial Heap

A Binomial Heap [19] is a type of heap that unlike the Binary Heap, which takes on the form of a Binary Tree, takes on the form of a collection of Binomial Trees. But much like the Binary Heap, the Binomial Heap must also satisfy the heap property constraint. In a binomial heap, there is at most one Binomial Tree of a given degree. Because there can only be one Binomial Tree of a given degree in a Binomial Heap, and the number of nodes in a Binomial Tree of degree k is 2k, the binary representation of the number of nodes in the heap will tell us which degree of Binomial Trees are present in the heap. For example, a Binomial Heap with 9 nodes gives us 9 = 1001 indicating that one tree of degree 3 and one of degree 0 in collection. The binary representation of the number of nodes in the Binomial Heap gives great insight into the structure of the heap. This can also be used to understand the operations on a Binomial Heap. Inserting into a Binomial Tree works just like the addition of binary numbers. Inserting a new node into the Binomial Heap with 9 nodes equates to 1001+0001 = 1010 = 10, which is the number of nodes after insertion. Every time to carry the 1 a merge of two Binomial Trees is performed, when performing this merge it's important to make sure that the heap property is not violated.

Binomial Heaps are typically explicitly defined, but there are ways to implicitly define them. However such implicit definitions do not reduce the overhead as drastically as it does with the Binary Heap if care about maintaining the worst-case costs. One such implementation where the Binomial Heap is implicitly defined as a contiguous array practically requires lg*n extra pointers and some extra information to be stored compared to the implicit definition of the Binary Heap [20].

2.18 Fibonacci Heap

A Fibonacci Heap [21] is a forest of rooted trees, each fulfilling the heap property constraint. Always keep track of which of the trees in forest has the most extreme value, min or max depending on requirement. Theoretically, the Fibonacci Heap is really fast compared to the other heaps, especially if the extract-min operations are few compared to the other operations. However, practically the large overhead of the Fibonacci Heap makes it less desirable for most applications.

Unlike the typical implementation of the Binomial Heap, which is also based on a forest of trees existing in a root list, do not care about having only one tree of each degree at all times. This allows for inserting any item as a tree of degree 0 which can be done in $O(1)$ time. Instead, only care about this after performs the pop-min operation, after performing pop-min, merge trees, allowing at most one of a given degree. A similar approach is taken to updates which are particularly required. If the key of a node is updated such that it violates the heap property that node is removed from the sub-heap and moved to the root list, and its parent is marked a loser. If a parent node that is marked a loser loses yet another node, then the parent node is also moved to the root list. This is done to avoid a situation where a shallow tree with a lot of nodes since extracting the minimum value would then be expensive. This behavior is often referred to as lazy and is based on the simple idea that doing work in bulk is often more efficient.

2.19 Index Sequential Priority Queue

To implement Index sequential priority queue [22] abstract data type in such a way that priority is set as an index so that it maintains the order as well as sequentially inserts and deletes a list of elements. Used of this approach removes the constraint of linked list and heap. In this approach, set the size of the index (list) as per minimum and maximum priority. For each priority, the index inserts a list of elements without shifting any element with their respective position. There is a rear pointer that shows the end of each of the priority queues. There is a front pointer that deletes an element from the queues sequentially and according to the priority. Duplicate priority elements are also placed at the same index position sequentially.

The execution time needed for an insertion and deletion operation is $O(1)$ time. An advantage of this algorithm is to save time on swapping as well as on splitting the link while the insertion and deletion of nodes occur. The length of the same priority node is independent of the actual length of the queue.

This algorithm presented scalability according to time and space as well as removing the task of exchanging the element according to the precedence of an element in a queue by using index sequencing. This algorithm minimizes the operation of the heap and linked list by exploring the index mechanism as a priority to implement the priority queue efficiently.

3. APPLICATION

The following table 1[23] shows the field and application where Binary tree utilized.

Table 1. Application of Binary Tree.

Field	Application
File System	Organizing and storing files.
Search Engines	Indexing and ranging web pages.
Sorting	Binary insertion and quick sort algorithms.
Math	Representing math expressions for evaluating and simplifying.
Data Compression	Encoding character using Huffman coding
Decision Trees	Modeling decisions and consequences in machine learning.
String Retrieval	Storing and retrieving string using trie data structure.
Game AI	Modeling Possible moves and outcomes in game artificial intelligence.
Network Routing	Routing data through computer network.
Arithmetic coding	Encoding character with variable length codes using binary tree.
Priority Queues	Efficient access to the item with the highest priority.
Image Processing	Representing image regions and shapes
Cryptography	Generating and managing encryption and decryption keys.

4. CONCLUSION

Although the list of research is not anticipated to be exhaustive, this study elaborates on research connected to several Binary tree Algorithms that are used to create priority queues. Despite this, the algorithm can be built very efficiently in terms of execution. Hybridization and split it to make more efficient. According to research, there are numerous more processes needed to meet the requirements for either storage space or execution time**.**

5. REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools" (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA, (2006).

[2] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. (2008) "Memory Systems: Cache, DRAM", Disk. Morgan Kaufmann.

[3] Adel'son-Vel'skii, G. M., and Landis, E. M. (1962), "An Algorithm for the Organization of information", Accession number: AD0406009.

[4] Michael T. Goodrich, Roberto Tamassia, and David M. Mount, "Data Structures and Algorithms in C++", John Wiley & Sons, Inc, ISBN: 0-471-20208-8.

[5] Michael. T. Goodrich, Roberto Tamassia, "Algorithm Design Foundations, Analysis, and Internet Examples". John Wiley & Sons, Inc. ISBN: 0-471-38365-1.

[6] Robert L. Kruse Alexander J. Ryba, "Data Structures and Program Design in C++", Prentice Hall, ISBN: 0-13-768995-0, 1998.

[7] W. A. Martin and D. N. Ness (1972) "Optimizing Binary Search Trees Grown with a Sorting Algorithm" In Communication of ACM, Vol. 15, Issue. 2, pp. 88-93.

[8] Day, A. C., 1976, "Balancing a Binary Tree", Computer Journal, XIX, pp. 360-361.

[9] Mark Allen Weiss, "Data Structure and Problem Solving using Java 3rd Edition" Addison-Wesley, ISBN: 9780321322135.

[10] His Chang and Sitharama Iyengar (1984) "Efficient Algorithms to Globally Balance a Binary Search Tree", In Communication of ACM, Vol. 27, Issue.7, pp. 695-702.

[11] Daniel Dominic Sleator and Robert Endre Tarjan (1985) "Self Adjusting Binary Search Trees", In Journal of the Association for Computing Machinery, Vol. 32, No. 3.

[12] William Stallings, "Computer Organization and Architecture" 7th Edition, Pearson Edition, ISBN: 81-7758-993-8.

[13] Quentin F. Stout and Belle L. Warren (1986) "Tree Rebalancing in Optimal Time and Space", In Communications of the ACM, Vol. 29, No. 9.

[14] A. Andersson. (1993) "Balanced Search Trees Made Simple", WADS.

[15] J. W. J. Williams (1964) "Algorithm 232: Heapsort", Communications of the ACM 7, 6, pp. 347-348.

[16] Donald B. Johnson. (1975) "Priority queues with an update and finding minimum spanning trees". Information Processing Letters, 4(3):53-57.

[17] Anthony LaMarca and Richard Ladner. (1996) "The influence of caches on the performance of heaps". Journal of Experimental Algorithmics, 1:4.

[18] Seidel, Raimund and Aragon, Cecilia R. (1996) "Randomized Search Trees", Algorithmica 16 (4/5): pp. 464-497.

[19] Vuillemin, Jean: (1978) "A data structure for manipulating priority queues". Communications of the ACM. 21 (4), pp. 309–315.

[20] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. (1988) "An implicit binomial queue with constant insertion time". In Rolf Karlsson and Andrzej Lingas, editors, SWAT 88: 1st Scandinavian Workshop on Algorithm Theory Halmstad, Sweden, Springer Berlin Heidelberg. LNCS 318, pages 1-13.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 3rd edition,(2009).

[22] F. Amjherawala, U. Amjherawala. (2019) 'A New Algorithm to Implement Priority Queue with Index Sequential Chaining', Springer International Conference on Advanced Computing Networking (ICANI), Advances in Intelligent Systems and Computing 870, pp 421-428.

[23] Kumar A (2023) Application of Binary Tree. https://www.codingninjas.com. Accessed 11 August 2023.